Model-checking of Solidity Smart Contracts (2)

Sebti Mouelhi
August, 2021

1 Smart contracts

Some of the function post-conditions cannot be described under SOLC-VERIFY, only pseudo-formulas are provided. Remix - Ethereum IDE was used to automate code debugging.

1.1 `payAll` overflow vulnerability

Let's consider this extended version of function `payAll` included in a working contract `Token`.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.7.0;
contract Token {
    mapping(address=>uint) balances;

    // initialization function (for three users)
    function initilize() public {
        balances[0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2] = 100000; //from
        balances[0x583031D1113aD414F02576BD6afaBfb302140225] = 0;      //to[0]
        balances[0x617F2E2fD72FD9D5503197092aC168c91465E7f2] = 0;      //to[1]
    }
    function payAll(address from, address[] memory to, uint val) payable public {
        // the length of array "to" is assumed to be equal to 2
        require(to.length == 2);
        uint amount = val * to.length; // overflow vulnerability here
        // require()'s condition is intended to be satisfied by the
        // exploit to go through the code and wrongly modify the balances
        require(balances[from] >= amount);
        balances[from] -= amount;
        for (uint i = 0; i < to.length; i++) {
            balances[to[i]] += val;
        }
    }
}
```

After deployment and initialization, the balances content is set as follows.

```
0x583031d1113ad414f02576bd6afabfb302140225 => 0
0x617f2e2fd72fd9d5503197092ac168c91465e7f2 => 0
0xab8483f64d9c6d1ecf9b849ae677dd3315835cb2 => 100000
```

Let's now run a `payAll` transaction with the following arguments:

`from:` `0xAb..`

`to:` `[0x61.., 0x58..]`, these addresses are assigned to 0 in `balances`

`val:` $57896044618658097711785492504343953926634992332820282019728792003956564819969 = \frac{\texttt{type(uint).max}}{2} + 2$

The value `val` $* 2$ (`to.length` $= 2$) assigned to `amount` overflows its 256-bits `uint` size. The value effectively written in `amount` is 2. The `require()`'s condition is satisfied since `balance[from]` $= 100000$ is greater than `amount` $= 2$. At the end of the transaction, the balance state is the following.

```
0xab8483f64d9c6d1ecf9b849ae677dd3315835cb2: 99998 uint256
0x617f2e2fd72fd9d5503197092ac168c91465e7f2:
 57896044618658097711785492504343953926634992332820282019728792003956564819969 uint256
0x583031d1113ad414f02576bd6afabfb302140225:
 57896044618658097711785492504343953926634992332820282019728792003956564819969 uint256
```

This vulnerability could be exploited to credit the attackers' balances **with fake generated amounts**. The library `SafeIntMath` could be used to prevent such arithmetic overflow exploits (see Notes 1).

1.2 `payAll` post-conditions

If the following two conditions are not assumed to hold at the function entry-point,

- $(\forall \texttt{i})\ \texttt{from} \neq \texttt{to[i]}$
- $(\forall \texttt{i}, \texttt{j})\ \texttt{i} \neq \texttt{j} \Rightarrow \texttt{to[i]} \neq \texttt{to[j]}$

the post-conditions could be reformulated (`b` is `balances` for short):

- $\texttt{b[from]} = \texttt{old(b[from])} - \texttt{val} \times (\texttt{to.length} - \texttt{occs(from, to)})$
- $(\forall \texttt{i})\ \texttt{to[i]} \neq \texttt{from} \Rightarrow \texttt{b[to[i]]} = \texttt{old(b[to[i]])} + \texttt{val} \times \texttt{occs(to[i], to)}$

with $\texttt{occs}(\texttt{v}, \texttt{tab})$ is a function returning the number `v` occurrences in the array `tab`.

1.3 `transfert` post-conditions

```
function transfer(address from, address to, uint val) public {
    uint updatedFrom;
    uint updatedTo;
    if (balances[from] >= val) {
        updatedFrom = balances[from] - val;
        updatedTo = balances[to] + val;
    }
    else {
        revert();
    }
    balances[from] = updatedFrom;
    balances[to] = updatedTo;
}
```

The exhaustive list of `transfert()`'s post-conditions is the following (`b` is `balances` for short).

1. $(\texttt{val} \leq \texttt{old(b[from])} \wedge \texttt{from} \neq \texttt{to}) \Rightarrow \texttt{b[from]} = \texttt{old(b[from])} - \texttt{val}$
2. $(\texttt{val} \leq \texttt{old(b[from])} \wedge \texttt{from} \neq \texttt{to}) \Rightarrow \texttt{b[to]} = \texttt{old(b[to])} + \texttt{val}$
3. $(\texttt{val} \leq \texttt{old(b[from])} \wedge \texttt{from} = \texttt{to}) \Rightarrow \texttt{b[from]} = \texttt{old(b[from])} + \texttt{val}$
4. $\texttt{val} > \texttt{old(b[from])} \Rightarrow \texttt{anything}$ (since the `revert()` is replaced by false under SOLC-VERIFY, see Notes 1)

The third post-condition shows the function's bug when `from` and `to` are the same. The function could be corrected as follows:

```
function transferCorrected(address from, address to, uint val) public {
    uint updatedFrom;
    uint updatedTo;
    if (balances[from] >= val) {
        if (to != from){
            updatedFrom = balances[from] - val;
            updatedTo = balances[to] + val;
        }
        else{
            updatedFrom = balances[from];
            updatedTo = balances[from];
        }
    } else {
        revert();
    }
    balances[from] = updatedFrom;
    balances[to] = updatedTo;
}
```

The third post-condition above could be reformulated as follows to depict the expected correct behavior:

$$(\texttt{val} \leq \texttt{old(b[from])} \wedge \texttt{from} = \texttt{to}) \Rightarrow \texttt{b[from]} = \texttt{old(b[from])}$$

2 Transmuter model

The updated optimized version of Transmuter presented in Notes 1 **is not equivalent to the naive version**. Given three users, the following is a counter-example trace defined as a sequence of public function calls (`accs`, `unex`, `exch`, `otick` are resp. `accounts`, `unexchanged`, `exchanged`, and `occupiedTick` for short).

```
deposit(100,0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2)
deposit(100,0x617F2E2fD72FD9D5503197092aC168c91465E7f2)
deposit(100,0x583031D1113aD414F02576BD6afaBfb302140225)
// get..() calls
exchange(5)
// get..() calls
deposit(100,0x583031D1113aD414F02576BD6afaBfb302140225)
// get..() calls
```

`TransmuterSpec.sol`	`TransmuterOptSpec.sol`
Trace	

`deposit(100, 0xAb..)`
`deposit(100, 0x61..)`
`deposit(100, 0x58..)`
	`deposit(100, 0xAb..)`
	`deposit(100, 0x61..)`
	`deposit(100, 0x58..)`

State variables	

`accs[0xAb..] = (unex:100, exch:0)`
`accs[0x61..] = (unex:100, exch:0)`
`accs[0x58..] = (unex:100, exch:0)`
	`[getUnex,getExch](0xAb..) = [100, 0]`
	`[getUnex,getExch](0x61..) = [100, 0]`
	`[getUnex,getExch](0x58..) = [100, 0]`
	$(\forall$`owner`$)$ `accs[owner].otick` $=1$
	`ticks[0] = (unex:0, rate: 0)`
	`ticks[1] = (unex:300, rate: 0)`
	`currTick = 1`

Trace	

`exchange(5):`
`delta` $=\frac{5}{300}(100+0)=1.66666666667$ (the same $\forall$ owners)
$(\forall$`owner`$)$ `accs[owner].unex` $\geq$ `delta`
	`exchange(5):`
	`currTick = 2, ticks[currTick = 2].rate = 0.0166..`
	$(\forall$`tick`$)$ `ticks[currTick].rate` $-$ `ticks[tick].rate` <1

State variables	

`accs[0xAb..] = (unex:98.33.., exch:1.66..)`
`accs[0x61..] = (unex:98.33.., exch:1.66..)`
`accs[0x58..] = (unex:98.33.., exch:1.66..)`
	`[getUnex,getExch](0xAb..) = [98.33.., 1.66..]`
	`[getUnex,getExch](0x61..) = [98.33.., 1.66..]`
	`[getUnex,getExch](0x58..) = [98.33.., 1.66..]`
	$(\forall$`owner`$)$ `accs[owner].otick` $=1$
	`ticks[0] = (unex:0, rate: 0)`
	`ticks[1] = (unex:300, rate: 0)`
	`ticks[2] = (unex:0, rate: 0.0166..)`
	`currTick = 2`

Trace	

`deposit(100, 0x58..):`
`accs[0x58..].unex = 100 + 98.33.. = 198.33..`
	`deposit(100, 0x58..):`
	`migrate(0x58..):`
	`ticks[accs[0x58..].otick].unex = 300 − 100 = 200`
	`accs[0x58..].unex = 98.33..`
	`accs[0x58..].exch = 1.638.. = 98.33.. × 0.0166..`
	`accs[0x58..].otick = 2`
	`ticks[2].unex = 98.33..`
	`accs[0x58..].unex = 198.33..`
	`ticks[2].unex = 198.33..`

State variables	

`accs[0xAb..] = (unex:98.33.., exch:1.66..)`
`accs[0x61..] = (unex:98.33.., exch:1.66..)`
`accs[0x58..] = (unex:198.33.., exch:1.66..)`
	`[getUnex,getExch](0xAb..) = [98.33.., 1.66..]`
	`[getUnex,getExch](0x61..) = [98.33.., 1.66..]`
	`[getUnex,getExch](0x58..) = [198.33.., 1.638..]`
	`ticks[0] = (unex:0, rate:0)`
	`ticks[1] = (unex:200, rate:0)`
	`ticks[2] = (unex:198.33.., rate:0.0166..)`

According to the trace above, after the second `deposit(100, 0x58..)`, the `.exchanged` attribute computed by `TransmuterSpec.sol` is different from that computed by `TransmuterOptSpec.sol`. In the first, the direct account balances updates (of attributes `.unexchanged` and `.exchanged`) are distributed between the function `exchange()` and `deposit()/withdraw()/claim()`. In the second, balances direct updates are **exclusively (but partially)** made by the `deposit()/withdraw()/claim()` functions. The `get..()` functions of the second **don't directly** return the current values of balances, they **`+/-` them to `newlyExchanged()` outputs** before returning.

This last detail is debatable, it seems possible to directly update the balances in a more enhanced/compact way within the `deposit()/withdraw()/claim()` and `get..()` functions in order to avoid having duplicate statements including `-/+ newlyExchanged(owner)` operations.

```
function migrate(address owner) internal {
    ...
    accounts[owner].unexchanged -= newlyExchanged(owner);
    accounts[owner].exchanged += newlyExchanged(owner);
    ...
}
```

The difference between the `.exchanged` values is due to the instructions above. In `TransmuterOptSpec.sol`, the new value of `accounts[owner].exchanged` is computed using the `.unexchanged` value newly updated just before, and this is due to calling `newlyExchanged(owner)` in the assignment operation `+=`. However, the balances `.exchanged` and `.unexchanged` are independently computed in `TransmuterSpec.sol`, and are exclusively based on the rate `delta` in the `exchange()` function computed just before the updates.

```
for account in accounts {
    // delta is computed once before the updates
    number delta = fraction * userStaked(account);
    if (account.unexchanged >= delta) {
        // independently computed based on delta
        account.unexchanged -= delta;
        account.exchanged += delta;
    }
    else {
        totalBuffered += delta - account.unexchanged;
        account.exchanged += account.unexchanged;
        account.unexchanged = 0;
    }
}
```

`accounts[owner].occupiedTick` being initialized to 0

It seems a priori that `accounts[owner].occupiedTick` being initialized to 0 does not cause any problem in the contract `TransmuterOptSpec.sol`. All the four statements using the `accounts[owner].occupiedTick` index in the contract are indicated below.

```
function migrate(address owner) internal {
    ticks[accounts[owner].occupiedTick].unexchanged -= accounts[owner].unexchanged; // 1
    accounts[owner].unexchanged -= newlyExchanged(owner);
    accounts[owner].exchanged += newlyExchanged(owner);
    accounts[owner].occupiedTick = currentTick; // 2
    ticks[currentTick].unexchanged += accounts[owner].unexchanged;
}
function newlyExchanged(address owner) internal {
    if (ticks[accounts[owner].occupiedTick].unexchanged == 0) { // 3
        return accounts[owner].unexchanged;
    }
    else {
        number rate = ticks[currentTick].rate - ticks[accounts[owner].occupiedTick].rate; // 4
        return accounts[owner].unexchanged * rate;
    }
}
```

In the first `// 1`, `accounts[owner].occupiedTick = 0` when `migrate()` is called for the first time in `deposit()`, `withdraw()` or `claim()` for a given `owner`. Since `ticks[accounts[owner].occupiedTick].unexchanged = 0` and `accounts[owner].unexchanged` is initially equal to 0, no problem occur in this case. The second `// 2` statement is an update of `accounts[owner].occupiedTick` to an index `t` > 0. In the third `// 3`, there are two cases: i) the first is when `newlyExcnaged(owner)` is called for the first time in `migrate(owner)`, the `if` control body is executed and returns `accounts[owner].unexchanged`, and ii) the second is when `newlyExchanged(owner)` is called by a `get..(owner)` function before calling `migrate()` for the first time in `deposit()/withdraw()/claim()` for a given `owner`, the `if` body is again executed and returns `accounts[owner].unexchanged`. The fourth `// 4` usage of the index `accounts[owner].occupiedTick` in the computation of `rate` is not reachable when `accounts[owner].occupiedTick` is equal to 0.