D3: Arithmetic Issues

Portland State
Computer Science

#3: Arithmetic Issues

- **Integer overflow** and **integer underflow**
- Unsigned vs signed integer confusion
- Turns many benign-seeming codepaths into vectors for theft or denial of service.

Walkthrough scenario

- A **contract**'s withdraw() function allows you to retrieve ether donated to the contract as long as your balance remains positive after the operation.
- An **attacker** attempts to withdraw more than his or her current balance.
- The check in withdraw() function is done with unsigned (positive) integers, resulting in an always positive condition.
- **Attacker** withdraws more than allowed and the resulting balance underflows and becomes orders of magnitude larger than it should be.

Example

- April 22, 2018
  - https://peckshield.com/2018/04/22/batchOverflow/

New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299)

ERC-20 Tokens Deposit Suspended

OKEx

April 24, 2018 22:09 · Created

Dear valued customers,

We are suspending the deposits of all ERC-20 tokens due to the discovery of a new smart contract bug - "BatchOverFlow". By exploiting the bug, attackers can generate an extremely large amount of tokens, and deposit them into a normal address. This makes many of the ERC-20 tokens vulnerable to price manipulations of the attackers.

Code vulnerability example #1

- Underflow Code Example

```
function withdraw(uint _amount) {
  require(balances[msg.sender] - _amount >= 0);
  msg.sender.transfer(_amount);
  balances[msg.sender] -= _amount;
}
```

- Using `uint` makes require statement useless (`uint` can never be < 0!
  - Attacker has 5 tokens and withdraws 6
  - Ends up with 2^{255}-1 tokens instead in balance
- What would make this code problematic?

```
function popArrayOfThings() {
  require(arrayOfThings.length >= 0);
  arrayOfThings.length--;
}
```

  - Brick a contract by popping it when its length is 0

Code vulnerability example #2

- Overflow Code Example
  - Code seeks to send each address in `_receivers`, a certain `_value` amount of ETH from their account (`balances[msg.sender]`)

```
255   function batchTransfer(address[] _receivers, uint256 _value) public whenNotPaused returns (bool) {
256     uint cnt = _receivers.length;
257     uint256 amount = uint256(cnt) * _value;
258     require(cnt > 0 && cnt <= 20);
259     require(_value > 0 && balances[msg.sender] >= amount);
260
261     balances[msg.sender] = balances[msg.sender].sub(amount);
262     for (uint i = 0; i < cnt; i++) {
263         balances[_receivers[i]] = balances[_receivers[i]].add(_value);
264         Transfer(msg.sender, _receivers[i], _value);
265     }
266     return true;
267   }
268 }
```

- Line 257, the `amount` local variable is calculated as the product of `cnt` (the number of receivers) and `_value` (the amount to send each receiver)
- Line 258 ensures there are only 1-20 receivers
- Line 259 ensures the amount in our `balances` is more than the `amount`
- Line 261 updates our `balances`
- Line 263 updates the `balances` for each of the `_receivers`
- Any errors here?

- Contract Exploit
  - Pass two `_receivers` into `batchTransfer()`
  - Pass 2^{255} for `_value` (an arbitrary 256 bit integer)
- What is the value of `amount`?
- Do the checks in lines 258-259 pass?
- What is the effect of line 261?
- What happens in line 263 to the balance of each of the two receivers?
  - Receivers get an extremely large `_value` added to their accounts without costing a dime in the attacker's pocket!

```
255   function batchTransfer(address[] _receivers, uint256 _value) public whenNotPaused returns (bool) {
256     uint cnt = _receivers.length;
257     uint256 amount = uint256(cnt) * _value;
258     require(cnt > 0 && cnt <= 20);
259     require(_value > 0 && balances[msg.sender] >= amount);
260
261     balances[msg.sender] = balances[msg.sender].sub(amount);
262     for (uint i = 0; i < cnt; i++) {
263         balances[_receivers[i]] = balances[_receivers[i]].add(_value);
264         Transfer(msg.sender, _receivers[i], _value);
265     }
266     return true;
267   }
268 }
```

Remediation

- **Validation:** validate all arithmetic operations

```
contract Overflow {
  uint private sellerBalance = 0;
  function unsafe_add(uint value) returns (bool) {
    sellerBalance += value; // possible overflow
  }

  function safe_add(uint value) returns (bool ){
    require(value + sellerBalance >= sellerBalance);
    sellerBalance += value;
  }
}
```

- Using SafeMath library (or an equivalent)
  - https://ethereumdev.io/safemath-protect-overflows/

```
library SafeMath {
  function mul(uint256 a, uint256 b) internal constant returns (uint256) {
    uint256 c = a * b;
    assert(a == 0 || c / a == b);
    return c;
  }

  function div(uint256 a, uint256 b) internal constant returns (uint256) {
    // Note: Solidity automatically throws when dividing by 0
    uint256 c = a / b;
    return c;
  }

  function sub(uint256 a, uint256 b) internal constant returns (uint256) {
    assert(b <= a);
    return a - b;
  }

  function add(uint256 a, uint256 b) internal constant returns (uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
  }
}
```

- Replacing native opeartors with SafeMath in contracts

```
contract MyContract {

    using SafeMath for uint256;

    uint256 result;

    function MyAdd(uint256 a, uint256 b) {
      result = 0;
      result = a.add(b);
    }
}
```

SI CTF Lab 3.3: D3_TokenSale