Homework 3: Breaking Smart Contracts

Patrick McCorry

Kings College London, UK
patrick.mccorry@kcl.ac.uk

Abstract. We'll focus on how to build and break self-enforcing smart contracts. Remember, they are typically around 40 lines of code, but they hold millions of dollars of asserts. It is recommended to make notes on this homework sheet for future use.

1 Why do we care?

In 1997, Nick Szabo published a blog about the idea of a *smart contract* where contractual clauses can be embedded in software. He envisioned a future where *all sorts of property that is valuable* will be controlled by digital means:

> *Smart contracts reference that property in a dynamic, proactively enforced form, and provide much better observation and verification where proactive measures must fall short. And where the vending machine, like electronic mail, implements an asynchronous protocol between the vending company and the customer, some smart contracts entail multiple synchronous steps between two or more parties.* - Nick Szabo

For awhile (and sometimes still today), it is often claimed that lawyers and coders will work together to write smart contracts. While this may be the case for some applications, it is in fact a restrictive view to think that smart contracts only deal with legal issues. As we have learnt in class, smart contracts are publicly verifiable and programmatic third parties who solve the *co-ordination problem*. They oversee (and self-enforce) a protocol's execution whenever there is no appointable third party:

> To be clear, at this point I quite regret adopting the term *smart contracts*. I should have called them something more boring and technical, perhaps something like *persistent scripts*. - Vitalik Buterin[1]

Over the years, we have witnessed the deployment of high-profile smart contracts holding millions of dollars of assets including the Ethereum Name Service (160k+ eth), WrappedEther (2.1m+ eth) and DigixDAO_MultiSig (395k+ eth). At the same time, we have also witnessed pseudonymous attackers exploit bugs in contracts to drain (or freeze) their coins including TheDAO (3.6m+ eth) and

[1] https://bitcoinist.com/vitalik-buterin-ethereum-regret-smart-contracts/

Parity's multi-signature contract (514k eth). Before you can build smart contracts in the wild, you must learn how to break them.

In this homework, we'll focus on general information about smart contracts before deep-diving into some attacks. The coding aspect will be covered during the lab session, and instead we'll focus on the theory.

2 Understanding smart contracts

Let's focus on the background information about smart contracts (and their capabilities).

- What is a smart contract? [2 marks]
- A program that runs on the blockchain. Acceptable: programmatic third party.
- Name and explain two types of accounts on Ethereum. [4 marks]
- **Externally owned account** This is a private-public key pair. Owned by a user.
  Contract account Code and storage for the program..
- Name and explain three types of transactions in Ethereum [6 marks]
- **Value transaction** Send coins to another account. No program execution/deployment.
  Creation Transaction Deploys a smart contract to the network.
  Invocation Transaction. User can call a function in a smart contract.
- What is gas? And how do we set its price? [4 marks]
- Gas is a standard metric to measure computation (and storage) on the network. The user can set the price they are willing to pay using a transaction.
- At a high level, what is the compilation process for Solidity before it can be deployed to the network? [3 marks]
- The source code is compiled to opcodes which is compiled to bytecode.

2.1 Breaking smart contracts

As we have mentioned, pseudonymous attackers (think devops199) have taken down (and drained) smart contracts holding millions of dollars of assets. Let's consider some of the classic examples.

- What is a front-running attack for smart contracts and how can an average user do it without collaborating with a miner? Please explain with an example. [6 marks]
- A front-running attack requires the attacker to publish a transaction in advance that changes the execution of an honest party's transaction. For example, if the user thinks they will withdraw some coins from a smart contract, the adversary can broadcast a transaction that changes the contract's state such that the coins are no longer available. Typically, miners include the highest fee paying transactions first, so an adversary just needs to pay a higher fee than an honest user.

- What is the checks-interaction-effects paradigm? Please provide an example. [3 marks]
- When coding a function, we should always check the pre-conditions are satisfied (i.e. user has sufficient balance to withdraw),then we should update the local state (i.e. remove the user's balance), and then interact with other accounts/contracts (i.e. send the balance).
- Why is a contract-in-the-middle attack possible if `tx.origin` is used instead of `msg.sender`? [4 marks]
- `tx.origin` identifies the transaction signer and `msg.sender` identifies the immediate caller. If contract B relies on `tx.origin` then it cannot distinguish if the immediate caller is the transaction signer or a malice contract. (i.e. malice contract can trick a user to call it, and then it calls contract B to steal masquerade as the user). If contract B relies on `msg.sender`, then it will always distinguish the immediate caller's address.
- What is an out-of-gas exception and how can it be used to lock up coins? [4 marks]
- Whenever a transaction runs out of gas (i.e. it has used up all purchased gas), then all computation will be reverted as if the transaction never happened. If there is an expensive computation in the contract before it sends coins to a user, then the expensive computation may prevent the transaction ever reaching the code that sends the coins.

```
contract NotAPonzi {

  function playGame() public payable {
    require(msg.value== 1 ether); // 1 coin deposit

    // Contract has 10 eth?
    if(address(this).balance == 10 ether) {
      msg.sender.transfer(address(this).balance);
    }
  }
}
```

Fig. 1: A broken contract

- Figure 1 outlines a contract that is vulnerable. How can a malice contract use `selfdestruct` to attack it? [4 marks]
- The command `selfdestruct` will always send coins to the appointed contract, even if that contract has no payable function. Thus an adversary can set up a malice contract with 1 wei, and then selfdestruct it with the intention to send 1 wei to the victim contract. Since its balance has 1 wei, it may never satisfy the condition (i..e balance == 10 eth) for coins to be sent.

- At a high level, how does a re-entrancy attack work? And how can we prevent it? [6 marks]
- A re-entrancy attack requires the victim contract to send coins before updating its internal state. This lets an adversary contract re-call the victim contract's withdrawal contract several times via its fallback function. After the attack, several withdrawals will have happened, but the balance is potentially only deducted once. To prevent the attack, we can either use the checks-effects-interaction paradigm or reduce the gas allocation for the receiving contract (i.e. transfer/send function).

3 Advanced Contracts

We hope you enjoyed the attack section! By the end of this class, you'll be a master of adversarial thinking! Let's take a break from attacks, and focus on some advanced smart contract features.

- What is an oracle smart contract? And why do we need it? [4 marks]
- An oracle smart contract verifies information from the external world and stores it for later use. It is required as a smart contract cannot interact with the external world (i.e. pull information) as this would prevent users deterministically validating the blockchain at some point in the future.
- What is a smart contract factory? And why is it useful to let another smart contract verify the code deployed? [4 marks]
- A factory lets us re-deploy different instances of a smart contract using the same code. It is useful for a smart contract to use the factory as it is confident that the instance matches the expected code. Otherwise, it can only provided an *address* of a pre-deployed contract and there is no guarantee what code it is actually executing.
- How can we configure smart contracts to support updates in the future? [4 marks]
- We can use a proxy contract that is responsible for accepting messages from contract A and forwarding it to contract B (and vice versa). Contract B can be the library with all the functionality and contract A can store the information. If we want to replace the library/contract B, we can update the proxy contract with the address of its replacement. Thus in the future all messages from contract A are sent to the replacement contract (and not contract B). But this approach so far requires human over-sight to perform the upgrade.
- Name and explain three permission systems that can be built into smart contracts. [6 marks]
- **Onlyowner.** The contract can have a single owner who can call special functions. Typically relies on the onlyOwner modifier.
  Multi-signature. A function cannot be called unless it has received a signature from k out of n parties.
  Pre-conditions A function may only be ready to call after a list of preconditions are satisfied. i.e. a vote function only works during the voting phase of a voting contract.

While the answers will be released in a week's time, if you found any of the questions difficult then please visit Patrick McCorry during his office hours.