Writing Smart Smart Contracts

Chase Troxell
Comp 116 Security

Abstract Over the past ten years, cryptocurrency has indelibly changed the way we think about money and value. The idea of a currency that is both unbacked by a central government, but rather, by cryptography is still seen as an area of confusion by many. However, it has undeniably gained prominence in the mainstream in many regards; Japan's largest banks, for example, have backed the cryptocurrency exchange, bitflyer.com, and even American investment firms, such as Goldman Sachs, have begun exploring cryptocurrency. As Bitcoin has grown more popular, so has the blockchain technology that it is built on. Enter Ethereum: a "decentralized platform that runs smart contracts."[1] In addition to representing a store of value via its native currency, ether, Ethereum leverages the blockchain to execute programs, called smart contracts. These smart contracts can be used to represent agreements between two parties, create new cryptocurrency tokens, or build entire decentralized applications. Because these programs are spread across the Ethereum blockchain, the resilience of the network can be leveraged to prevent censorship, guarantee availability and execute the programs in a trust-less environment. However, deploying smart contracts, particularly contracts related to stores of value, does not come without significant risks. In this paper, I will explore the vulnerabilities inherent to deploying user-written Smart Contracts into a hostile environment using a consensus-based blockchain.

[1] "Ethereum Project." *Ethereum Project*, Ethereum Foundation, ethereum.org/.

Table of Contents

1 Introduction

Since 1958, Americans have expressed greater distrust in government than ever before, creating the longest sustained period of low public trust in government in 50 years.[2] This distrust stems from concerns about how citizens can possibly be represented by a centralized authority, like politicians in a federal government, who are conceived as placing self interests ahead of their constituents. This distrust of centralized authority has also spread to encompass banking, following the 2008 recession, and has recently come to encompass large companies, like Google and Facebook over privacy concerns.

In light of this heightened distrust, it follows, then, that technologies would be designed to subvert centralized authorities through a decentralized model. Rather than employing institutions as a mediators, these technologies favor cryptographic evidence and consensus protocols to construct trustless peer-to-peer interactions. The inception of Bitcoin and the blockchain provide users with greater transparency by virtue of the fact that the technology is open source and the blockchain itself is browsable as a publicly verifiable record.

Blockchain technology has since been extended further to its use in Ethereum, in which the blockchain is used not only as a ledger, but as a Turing complete computer. This programmable platform represents one of the forerunners in the field of decentralized programming; however, for Ethereum to be successful, it must be vigilant of the security risks that its greatest boon represents. Where hackers might obtain client information (credit card, social security, etc) which could previously be exchanged for money, adversarial actors in this space stand to directly and irreversibly acquire money. In light of this, security must be given primary importance before deploying a public smart contract. In this paper, I will first present

[2] Pew Research Center, November, 2015, “Beyond Distrust: How Americans View Their Government ”

Bitcoin as a basic case study in use of the blockchain as a public ledger. From there, I will describe how Ethereum builds on this concept, extending it into a decentralized programming platform. I will then provide a brief tutorial on programming an Ethereum Smart Contract and describe security flaws within my example smart contract. Finally, I will consider the state of security in Ethereum and its implications going forward.

2 To The Community

Blockchain technology is constructed based on a series of consensus protocols which allow for globally verifiable transaction to take place. These protocols dictate how users interact with one another and how value and information is transferred to maintain a trustless consensus. However, Smart Contracts in the Ethereum blockchain bring an entirely new layer into place; the user created contracts operating atop the Ethereum protocol are easily confused for being as secure as the blockchain itself. Due to vulnerabilities in user contracts, attackers have been able to steal more than 200,000 ETH since the technologies. The problem comes about due to poor documentation and difficult to understand tools. Many insecure smart contracts are scattered within the documentation, as well as in many research papers and StackExchange posts. Further, Solidity itself implements many seemingly simple features in counter-intuitive ways and does not include means of handling blockchain specific phenomena, like computation delay. As such, it is imperative for anyone interested in pursuing Smart Contract development to get a working foundation in the protocol on which they're building and a solid understanding of the tools at their disposal.

3 Background

3.1 The Blockchain thru Bitcoin

One problem inherent in currency exchange is that of double spending; if one can simply copy their money and spend it many times, the currency quickly loses value and credibility. Most fiat currencies solve this problem via centralization, however, this leads to costly mediation, ultimately resulting in "limiting the minimum practical transaction size and cutting off the possibility for small casual transactions" and, on a broader scale, increasing the need for trust between parties due to the reversible nature of transactions.[3] Bitcoin, while not the first technology to attempt it, solved the double spending problem by constructing a blockchain based on proof of work. Transactions in this network are grouped together into blocks, each of which is verified and published publicly with a hash after a provable amount of work has been completed. Thus, the risk of double spending is mitigated by affording precedence to the earliest attempt to transact within a block of transactions.

The Bitcoin network is built to give precedence to the longest chain of verifiable blocks, thus, to reverse a transaction, one would need to redo the work of all following blocks. This mechanism limits the ability of an attacking party to alter past transactions because, to modify a block, "an attacker would have to redo the proof-of-work of the block … and then catch up with and surpass the work of the honest nodes".[4] Further, because blocks are incentivized, an attacker with this magnitude of CPU power would likely stand to gain more by participating honestly in the network. With these the protocol-level protections in place, the blockchain is used in Bitcoin

[3] Nakamoto, Satoshi. " A Peer-to-Peer Electronic Cash System." *Bitcoin.org*, 2008, bitcoin.org/bitcoin.pdf.

[4] Nakamoto, Satoshi.

as a public ledger to "record a public history of transactions" that are universally verifiable and allow the network to be flexible and trustless.[5]

3.2 Ethereum

Ethereum inherits many of the concepts that allow the blockchain to be a distributed, cryptographically secure ledger and leverages them in a different, more generalized way. In Ethereum, the blockchain is used as a means of obtaining distributed consensus, allowing "Ethereum [...] to provide a blockchain with a built-in fully fledged Turing-complete programming language that can be used to create "contracts" [...] to encode arbitrary state transition functions."[6] This programming language allows users to create smart contracts and build decentralized applications on top of the blockchain, with the use of its native currency, ether. In Ethereum, there are two types of accounts: externally owned accounts, which is controlled by a person or organization, and contract accounts, which are controlled by a program. Contracts, therefore, are more akin to autonomous agents; when they receive transactions, or messages, they can send new messages, write to their internal storage or even generate other contracts.

Transactions between accounts contain the recipient, the signature of the sender and data, as in any financial transaction. Ethereum also leverages the idea of gas, calculated in small amounts of Ether, to prevent denial of service attacks; essentially, gas limits the resultant computation of any transaction by pre-promising (and paying for) a number of execution steps. Messages are similar to transactions, however these are passed between contracts; an important

---

[5] Nakamoto, Satoshi.

[6] Buterin, Vitalik. "A Next-Generation Smart Contract and Decentralized Application Platform." 2014.

reservation here is that a message/transaction to a contract that generates sub-executions is limited by the initial gas provided. That is to say that given a transaction from A to Contract B with 10 gas, if B consumes 1 gas to call C, which consumes 6 gas, B is left with 3 gas to execute to completion.

Finally, contract code is run on the Ethereum Virtual Machine on all Ethereum network nodes. Code execution has access to a persistent storage, which exists in contract accounts, and s non-persistent memory and stack, which are used during execution. Code is executed in an infinite loop until a `STOP` or `RETURN` code is reached in the contract's instructions or the transaction's gas runs out. Contract code is stored in EVM code, which is a "low-level, stack-based bytecode language."[7] Generally, however, code for smart contracts is written in higher level languages, including Solidity and Viper. [8] [9]

4 Smart Contracts

4.1 Use Cases

Smart contracts have a potentially infinite variety of uses; any possible computation can be written and deployed to the blockchain. Applications that are built using Smart Contract code behind the scenes are commonly referred to as Distributed Applications, or Dapps for short. Ethereum lends itself to a few different classes of application, the first of which is financial. For example, it is possible to create sub currencies by extending the following Solidity core code.

```
mapping (address => uint) balances;

function send(address to, uint amount) {
    if (balances[msg.sender] >= amount) {
```

[7] Buterin, Vitalik. "A Next-Generation Smart Contract and Decentralized Application Platform." 2014.

[8] https://solidity.readthedocs.io/en/develop/

[9] https://github.com/ethereum/viper

```
                balances[msg.sender] -= amount;
                balances[to] += amount;
            }
        }
```
[10]

This code essentially describes a distributed ledger, `balances`, and, when instructed by a user, can transfer an input amount into another user's account. This is the basis for token systems, which are used now within Dapps to quantify work and usage. For example, in Golem, a platform allowing users to rent out processing time on their computers, the native token, GNT, is used to pay for computations. In Augur, a distributed predictions market, REP token holders are rewarded for correctly reporting the results of events and penalized for lying.[11] Functionally, as use of a token increases, its value will increase, incentivizing early adoption and platform loyalty

These tokens are initially distributed through an Initial Coin Offering, or ICO, contract. In an ICO, the group behind the distributed application auctions off a percentage of all tokens for their application as a means of crowdfunding and token distribution. As of September 23, 2017, ICOs have generated over $2.3 billion dollars in crowdfunded capital.[12] These funds are typically collected in ether or another cryptocurrency; as such, ICO contracts should be well audited to avoid loss of investor funds.

Decentralized Autonomous Organizations, or DAOs, are also a potential use case for Smart contracts. It is possible to codify elements of an organization like voting on proposals or as removing members from the leadership group based on a consensus of organization members. Further, distribution of stock or ownership of the company could be transparently transferred

[10] Based on example in Buterin (2014)

[11] Xie, Linda. "A Beginner's Guide to Ethereum Tokens." *Www.blog.coinbase.com*, 22 May 2017, www.blog.coinbase.com/a-beginners-guide-to-ethereum-tokens-fbd5611fe30b.

[12] Barnett, Chance. "Inside the Meteoric Rise of ICOs." *Forbes*, 23 Sept. 2017, www.forbes.com/sites/chancebarnett/2017/09/23/inside-the-meteoric-rise-of-icos/#42ed41405670.

between entities in a publicly verifiable way. A simple example of a DAO would be a multisignature wallet, in which the funds are unlocked when, for example, 5 of 7 signatories approve a proposed transaction. These are a few of the possible uses; there are many other applications from job boards to games built onto Ethereum now.

4.2 Developing Smart Contracts

At the time of writing (Dec 2017), there are a few competing standards for Smart Contract creation, testing and deployment. For the purposes of this paper, I will describe a development cycle using Ganache, Truffle and Solidity.

Solidity is a high level language for contract development that is "statically typed, supports inheritance, libraries and complex user-defined types."[13] It is fairly similar to Javascript and C++ and is compiled directly into EVM code for deployment to the blockchain. Ganache is a program to simulate a blockchain on your computer; it provides a local test environment and includes a simple blockchain explorer. Truffle is a development framework that encapsulates a pipeline for contract deployment and testing. This stack was used to develop the example contract in Section 5.

4.3 Smart Contract Structure in Solidity

Next, we will analyze a simple contract written in Solidity to get a basis of understanding on how contracts are programmed.

```
pragma solidity ^0.4.18;

contract SimpleStorage {

     address owner;
```

[13] "Solidity Documentation." *Solidity — Solidity 0.4.20 Documentation*, 2017, solidity.readthedocs.io/en/develop/.

```
        string storage;

        function SimpleStorage(string _status) {
             owner = msg.sender;
             storage = _status;
        }
        function getStatus() returns (string) {
             return storage;
        }
        function setStatus(string _status) {
             storage = _status;
        }
    }
```
[14]

In this contract, the first line `pragma solidity ^0.4.18;` directs the compiler to use any version of Solidity above 0.4.X, where X >= 18. Next, we declare the contract as SimpleStorage; SimpleStorage has two state variables, a string and an address type. A high level language, Solidity includes most primitives, including strings, signed and unsigned integers and arrays, as well as specific types, such as addresses, which are 20 byte references to Ethereum network addresses. Following this, there are 3 functions, a constructor, which sets the initial value of `storage` when the contract is first deployed to the blockchain. In addition, it sets the owner to the sender of the message to construct the contract, or the deployer. In the next two functions, the contract allows anyone to set the state variable, `storage`, and get its current state.

5 Sample Smart Contract and dApp

5.1 Installation and Use

To explore Smart Contract and Dapp development, I've created a simple application that leverages the Ethereum blockchain. The contract defines a wishlist of Books to which

[14] Example from "Hello World Ethereum: Create an Ethereum smart contract and interact with it from a website" by Yann-Aël Le Borgne @yannael on Github

contributors can send ether; after they've contributed, it is possible to rescind the contribution should one decide that the book doesn't meet his expectations. This application could be extended to a voting engine (i.e. the item with the most ether after a period wins) or to judge interest in a series of proposals and keep track of contributors. The code for this smart contract can be found at https://github.com/tuftsdev/comp116-ctroxell.git. It is deployed to the Ethereum blockchain on the Ropsten test network at address 0x0C63dF1945293a4c3950202536d61aeAe52eDf3a . I've also developed an HTML frontend to this application. To use the frontend:

1. Install Node.js and npm.
2. Run `git clone https://github.com/tuftsdev/comp116-ctroxell.git`
3. Run: `cd comp116/finalproject/ExampleDapp/frontend && npm install && http-server`
4. Install Metamask at **link**.
5. Set up Metamask and create 1 or more accounts.
6. Change your network to Ropsten testnet using the top left dropdown.
7. Click Buy ETH to request test Ether from the Ropsten faucet (ETH on the test network is free and has no value).
8. Navigate to localhost:8080 and try contributing/rescinding some ETH to a book.
    1. This should cause Metamask to ask for your approval; click Submit.

This application was purposefully built with a few common Solidity flaws; the following sections will go into depth on these flaws, as well as a few other common vulnerabilities. If you'd like to attempt to find the flaws in the contract code yourself, stop here. It is possible to deploy a test version of the contract using Remix IDE, a web based Solidity IDE available **here**. Simply copying and pasting the full contract code should suffice. Compiling and running the code will deploy it to the network; from there, finding the flaws in the program is left as an exercise to the reader.

A few hints: You will need to write and deploy at least two smart contracts; the EVM is stack based, therefore it will execute any function calls/message results before continuing a given function; fallback functions will help; simple erors can be disasteruous; Mirai has a friend here.

5.2 Sample Contract Vulnerability Enumeration

There were 3 security flaws included in the example:

1. Spelling error in BookWishlist's constructor: `BookWishlit()`. This error is easy to overlook; however, it's results could have been been disastrous. Because this function sets the owner of BookWishlist to be the function's invoker (normally the contract creator), this oversight could potentially give an attacker the ability to drain the contract of its Ether and destroy it, using the `kill()` function. This approach is used in the included contract, SpellingAttack.sol, to disable the targeted contract. Fortunately, the fix here is to simply correct the spelling, such that the constructor runs on contract creation as expected. The takeaway from this flaw is that if a modifier's condition (ie OnlyOwner) can be exploited via a flaw, the modifier is useless at best and, at worst, provides a false sense of security.
2. DOS potential in BookWishlist's `refundAll()` function. The offending line in this function is: `require(book.contributions[j].from.send(refund));` for a few reasons. The exploit I will describe is written in DOSAttack.sol. This attack functions by using a contract to contribute to a Book on the Wishlist. This contract's fallback function, which is triggered when it receives any ETH, holds an infinite loop which consumes gas, causing the refund transaction to fail. The other prong of this flaw is that the `send()` function in Solidity does not `throw()` when it fails, rather it simply returns false. Thus, should the Wishlist owner go to refund the contributors, it will eventually try to refund the

attacking contract. When an attempt to refund the attacker fails, `send()` returns false, triggering the `require()`, which will revert all previous refunds and cause execution to terminate. This attack would make it impossible for the contract owner to refund the contributions, potentially exposing the contract to infiltration for a longer period. To correct this, either use an if-statement to ignore failed refunds, or use `transfer()` to send refunds, as it throws an exception on failure.

3. Reentrancy potential in BookWishlist's `rescind()` function. This exploit also uses fallback function as the means of attack; however, in this case, a race condition in `rescind()` is targeted. The `rescind()` function checks if the requester has made a contribution, then sends the money and, finally, sets the requester's contribution to 0. The attack contract takes advantage of this by sending another `rescind()` in its fallback function; because the old contribution has not been set to 0, the second call (and the third, etc) will continue to grant the attacker more funds until the call runs out of gas or the contract runs out of funds. This bug can be avoided by using `transfer()`, which does not allow other contracts' fallback functions to run, or by using the pull-transfer paradigm, in which one checks the send condition, negates it in the contract's state and then sends the funds.

6 The State of the Art and Past Attacks

Since Ethereum's inception, there have been a number of attacks which have resulted in loss of significant amounts of Ether.

One widely publicized attack was on a DAO in June 2016; the DAO was a crowdfunding platform through which contributors could vote on different proposals to fund, similar to a VC firm. The vulnerability in this contact resulted in the loss of some $60M in Ether and a contentious hard fork to revert the losses.[15] This attack worked in a very similar way as the `rescind()` flaw in the BookWishlist contract; the developers included a withdrawal function which: A. allowed unknown code execution B. updated the balance of the withdrawer after sending ether and C. left underflow vulnerabilities (s.t. withdrawing 1 ETH from 0 would result in a 2^256 -1 balance). The combination of these factors resulted in an enormous loss of funds and destabilized the network.

Another devastating attack was on the Parity wallet in June 2017. The wallet included a mechanism for setting ownership of the contract via a private initialization function. Unfortunately, it also included a raw `delegatecall(data)` in its fallback function, which allowed anyone access to private functions by sending data to the fallback function of Parity wallet contracts. This allowed attackers to gain control of over 150,000 ETH ($30M at the time), by simply withdrawing from their newly controlled wallets.[16] Later, the fix for this wallet hack opened up an exploit which allowed attackers to permanently freeze funds at will.

In both of these cases, simple programming constructs which appeared benign allowed attackers to obtain control of crucial contract elements. Although executing unknown code is an obvious security flaw, Solidity obfuscates the danger of these calls semantically through the `send()` and `call().value()` functions. In addition, there is a distinct danger in trusting

[15] Atzei, Nicola, et al. “A Survey of Attacks on Ethereum Smart Contracts.” *Cryptology EPrint Archive*, 2016, eprint.iacr.org/2016/1007.pdf.

[16] Palladino, Santiago. “The Parity Wallet Hack Explained.” *Zeppelin Blog*, 19 July 2017, blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7.

function modifiers (private, OnlyOwner); any flaw which allows an attacker access to the base condition of these modifiers completely invalidates them.

Ultimately, the root of these exploits is a “mismatches between [contracts’] intended behaviour and the actual one” and the difficulty of detection.[17] At the present, solutions to this disconnect are few and far between. One of the forerunners on Smart Contract static analysis is Oyente, a tool which analyzes contract byte code and generates a control flow graph.[18] This graph can be used to detect reentrancy potential, unknown code execution and exception disorders before the contracts are even run. Another tool translates EVM byte code or Solidity into the functional language F* to check specifically for exception disorders and third party code execution. Further, this “tool verifies that the two pieces of code [i.e. Solidity code and EVM bytecode] have equivalent behaviours”[19] to help close the gap between intention and effect. However, a large scale analysis of contracts deployed to the Ethereum blockchain found that vulnerabilities were “widespread” and that “~ 28% of the analyzed contracts potentially contain “exception disorder” vulnerabilities.”[20]

7 Conclusion

Ultimately, Ethereum and consensus programming are very young technologies. At present, the exponential growth in investment in these fields and the relatively primitive security guidelines are poorly matched. However, it is uniquely typical for developers to work directly

[17] See footnote 15

[18] Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: ACM CCS (2016), http://eprint.iacr.org/2016/633

[19] See footnote 15

[20] See footnote 18

with large stores of value; as such, they are hugely incentivized to stay up to date on means of keeping their customers funds safe. Despite the fact that the field has a long way to go in terms of establishing best practices, security researchers have a rare opportunity to explore and, ideally, shape the incipient and rapidly growing field of Smart Contract development.

References

Atzei, Nicola, et al. “A Survey of Attacks on Ethereum Smart Contracts.” *Cryptology EPrint Archive*, 2016, eprint.iacr.org/2016/1007.pdf.

Barnett, Chance. “Inside the Meteoric Rise of ICOs.” *Forbes*, 23 Sept. 2017, www.forbes.com/sites/chancebarnett/2017/09/23/inside-the-meteoric-rise-of-icos/#42ed41405670.

Buterin, Vitalik. “A Next-Generation Smart Contract and Decentralized Application Platform.” 2014.

“Ethereum Project.” *Ethereum Project*, Ethereum Foundation, ethereum.org/.

Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: ACM CCS (2016), http://eprint.iacr.org/2016/633

Nakamoto, Satoshi. “ A Peer-to-Peer Electronic Cash System.” *Bitcoin.org*, 2008, bitcoin.org/bitcoin.pdf.

Palladino, Santiago. “The Parity Wallet Hack Explained.” *Zeppelin Blog*, 19 July 2017, blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7.

“Solidity Documentation.” *Solidity — Solidity 0.4.20 Documentation*, 2017, solidity.readthedocs.io/en/develop/.

Xie, Linda. “A Beginner’s Guide to Ethereum Tokens.” *Blog.coinbase.com*, 22 May 2017, blog.coinbase.com/a-beginners-guide-to-ethereum-tokens-fbd5611fe30b.