Botract: abusing smart contracts and blockchain for botnet command and control

Omar Alibrahim*

ISC Department,
CLS,
Kuwait University,
Adailiya Campus,
P.O. Box 5969, Safat 13060, Kuwait
Email: omar.alibrahim@ku.edu.kw
*Corresponding author

Majid Malaika

omProtect LLC,
P.O. Box 27229, Washington,
DC 20038, USA
Email: mmalaika@omprotect.com

Abstract: This paper presents how smart contracts and blockchains can potentially be abused to create seemingly unassailable botnets. This involves publishing command and control (C2) logic in the form of smart contracts to the blockchain and then calling the functions of the smart contract for sending and receiving commands and keeping track of the state of bots. We call this technique Botract, derived by merging two words: bot and contract. In addition to describing how hackers can exploit smart contracts for C2, we also explain why it is difficult to disarm Botract, given the distributed nature of the blockchain and the persistent nature of smart contracts deployed on top of them. We then describe the architecture for deploying blockchain-based botnets and implement a proof-of-concept using isolated testnet environments. Our goal is to prove the feasibility of our approach, which we hope will create awareness among the community on the importance of auditing smart contracts on the blockchain and defending against these botnets before they become widespread.

Keywords: smart contract; blockchain; security; botnets; Ethereum.


Biographical notes: Omar Alibrahim is an Assistant Professor at the ISC Department at Kuwait University. His research interests include blockchain security, threat intelligence and malware analysis.
Introduction

Traditionally, botnets have used thousands of machines controlled by a central entity, a bot commander, through the infection of a common malware. Botnets are frequently used for DDoS attacks, but they can also be used to send spam, host criminal websites, and perform other activities.

Throughout their historical development, botnets have gone through a series of evolutions to increase their resiliency and robustness (Abu Rajab et al., 2006; Chen and Bridges, 2017; Langner, 2011; Shin et al., 2012). A primitive bot will come pre-configured with a list of domain names or IP addresses so that it can connect to its command and control (C2) server(s). Because this list is often static, firewalls and intrusion detection systems (IDS) can identify these illicit communications via network signatures (e.g., using Snort, https://www.snort.org). Furthermore, bots can also be detected by searching for binary patterns in their code, like anti-virus signatures or hardcoded strings, but thanks to the use of code obfuscation, packers (e.g., Themida, https://www.oreans.com/themida.php) have made bots more difficult to reverse-engineer. Furthermore, bots can bypass firewalls that rely on rudimentary filters for detection by consulting anonymous hidden services reachable via the onion router (Tor) (Dingledine et al., 2004) – and other darknet resources.

In addition to code obfuscation and anonymity networks, recently, botnets have adopted steganographic C2 channels on social media sites, exploiting both the covertness of the communication channel and the pervasiveness of social networks. For example, the use of image steganography on Instagram (e.g., Deutsch and Garrie, 2017) as a means of transport for C2 has made it more appealing for botnet developers than other techniques. Furthermore, botnet developers have also innovated with domain generation algorithms (Sood and Zeadally, 2016), and domain name system (DNS) fast-flux techniques (Yadav et al., 2012).

As for bot communication, botnets have gone through a series of evolutions to secure C2 channels. In the past, botnets like Agobot and Spybot simply utilised internet relay chat (IRC) protocol, a cleartext protocol that can be easily detected and blocked (Abu Rajab et al., 2006). In contrast, botnets today typically utilise encryption to secure their channels, and may also adopt custom protocols, secure socket layer (SSL) certificate pinning, and other hardening techniques.

Over the past few years, joint efforts led by government, law enforcement, and private sector organisations have disrupted and dismantled numerous botnets worldwide [e.g., Conficker (Shin et al., 2012)]. Furthermore, these efforts have led to the development of open-source intelligence tools (e.g., ThreatConnect, https://www.threatconnect.com/) used to unveil and takedown, or sinkhole, many C2 servers (Chen and Bridges, 2017).
1.1 New techniques for botnet development and communication

As a continuation of the development of botnet techniques, this paper proposes Botract, a novel technique for botnet development where hackers can exploit the blockchain and smart contracts to build their own botnets. This brings up many advantages not available with other forms of botnets:

1. **Distributed platform**: hosting botnet command and control (C2) servers on the blockchain makes them more resilient against takedown and sinkholing because the C2s are not hosted on a single server that authorities can control. Instead, they are deployed on a public platform like the blockchain, which is distributed in nature and can not be controlled by a single entity.

2. **Consistent ledger**: blockchain is a decentralised ledger technology that secures the integrity of transactions through digital signatures. Using the blockchain, a new block is added to the ledger permanently and cannot be modified. Botnet developers can exploit the fact that insertions to the blockchain are persistent by deploying their own smart contracts and then executing them on the blockchain. These contracts will continue and remain executable on the blockchain as long as they are not programmed to self-destruct.

3. **Fast bootstrap**: botnet C2s can be easily launched by creating smart contracts that are deployed quickly, and the time it takes for them to become publicly active only depends on the network synchronisation time. These contracts can be portrayed as templates that a bot commander can use to deploy a new botnet C2. Consequently, bot commanders do not need to invest in any infrastructure, and they can easily create a new instance within a short period of time.

4. **Programmability and extensibility**: smart contracts can be written as modular class functions and data (to represent state) that reside at a specific address on the blockchain. A botnet commander can write custom code using the smart contract language to define the logic behind its C2 communication, and this code can be easily updated through code commits with a new address on the blockchain, just like the commits of code repositories, to implement a different logic.

1.2 Exploiting blockchain and smart contract technologies

In this paper, we present Botract for botnet control via smart contract on the blockchain. Not only we propose an alternative method to support messaging and control of botnets, but we also point out the design issues of the underlying blockchain technology for deploying smart contracts, including implicit end-user trust, lack of code scrutiny, and absence of governance structure.

In the development of the internet, blockchain technology [e.g., Ethereum (Wood, 2014)] is designed to be an open platform for its users. Accordingly, users of the blockchain can freely interact to deploy contractual logic used by multiple stakeholders. This means that users can write their own code for smart contracts and deploy them to the blockchain permanently without any review. Consequently, as the blockchain increases in its adoption and proliferates in its number of users, the impact of this implicit end-user trust will eventually lead to the abuse of its inherent functionalities, a design issue of the blockchain smart contract model. This paper intends to point out
these overlooked issues and calls for the development of a governance model for the blockchain and smart contracts that meets security requirements.

1.3 Organisation

The organisation of this paper is as follows: We give an overview of preliminaries in Section 2. Section 3 discusses related work in the area of smart contracts and blockchain security. In Section 4, we present the design and architecture of Bottract. Section 5 presents the evaluation of Bottract. Finally, we conclude in Section 6.

2 Preliminaries

In this section, we give some background information about blockchain and smart contracts.

2.1 Blockchain

The invention of the blockchain has emerged in the past decade as the core component of the digital currency, Bitcoin (Nakamoto, 2008), serving as the public ledger of all transactions. Embraced for addressing the double-spending problem, i.e., a single token can be consumed twice, the blockchain can be used to prevent equivocations of false transactions.

2.1.1 Characteristics of the blockchain

A few characteristics that set blockchains as a radical development and a breakthrough in the digital world are:

- the distributed architecture of the building nodes of the blockchain
- resistance to modification of the recorded data in retrospect, a prominent feature that distinguishes blockchains from databases- this means that you can only append to them
- agreement protocol between the nodes of the blockchain network for every transaction to be committed
- programmability with smart contracts, which made blockchains a true breakthrough.

While the first-generation blockchains were designed to perform a small set of simple operations, techniques for transactions of currency-like tokens have been developed to allow blockchains to perform more complex operations via smart contracts defined in full-fledged programming languages. Smart contracts are described below and are the core technology we exploit to build botnets.
2.2 Smart contracts

Smart contracts are programs that implement the conditional logic of a contract, which is verified and enforced by the blockchain. In other words, they are intended to facilitate, verify, or enforce negotiation or performance between users of the blockchain. Proponents of smart contracts claim that many kinds of contractual clauses can be made partially or fully self-executing, self-enforcing, or both. The aim with smart contracts is to automate execution and reduce the transaction costs associated with enforcing them.

Smart contracts are typically written in an object-oriented language like Solidity (http://solidity.readthedocs.io/). Solidity is a high-level language whose syntax is similar to that of JavaScript, and it is designed to target the Ethereum virtual machine (EVM) (Wood, 2014). It is statically typed and supports inheritance, libraries, and complex user-defined types, among other features. Using Solidity, it is possible to create contracts for voting, crowdfunding, blind auctions, multi-signature wallets, and other applications.

While still undergoing research and development, smart contracts are not left without problems in privacy, security, and availability (Atzei et al., 2017). In this paper, we show how smart contracts can be exploited by running a command and control (C2) for botnets using a custom contract published on the Ethereum blockchain. Hopefully, this will raise awareness among the blockchain community leading them to propose defenses against such botnets and to think about a governance framework for the blockchain.

3 Related work

Beyond traceable cryptocurrency, smart contracts have enabled new applications for blockchains. For example, Pan et al. (2019) proposed an IoT framework based on blockchain and smart contracts that enforces rules to regulate and audit IoT devices. Bader et al. (2018) investigated how to implement car insurance policies based on smart contracts that increases cost efficiency and process reliability. In addition, Yeh et al. (2018) showed an elegant solution to mobile payments using smart contracts and blockchain with certificateless pairing-based cryptography. In the same vein, Hsiao et al. (2018) suggested exploiting the secret sharing scheme and homomorphic encryption with blockchain to implement a secure decentralised e-voting system.

While many of the works have focused on developing virtuous applications for smart contracts and blockchain, malicious applications like botnets can exploit the same technologies as well. For this reason, an analysis of smart contracts is required. In the literature, smart contracts have been shown to be exposed to severe vulnerabilities (Atzei et al., 2017; Buterin, 2016). To overcome them, formal specification and verification of Ethereum contracts have been proposed (Grishchenko et al., 2018; Hildenbrandt et al., 2018). Hirai (2017) defined a formal model to verify smart contracts and prove safety properties. In addition to formal verification of smart contracts, automated security tools based on symbolic execution have been developed to reveal code vulnerabilities in smart contracts. Examples include Oyente (Luu et al., 2016), Mythril (https://github.com/ConsenSys/mythril), Maian (Nikolić et al., 2018) and Securify (Tsankov et al., 2018). Furthermore, secure coding practices have also been proposed for smart contracts. For instance, Delmolino et al. (2016) documented various vulnerabilities developers may introduce while writing smart contracts and proposed methods on how to avoid common pitfalls. In general, the work in this area examines smart contracts for
their correctness of execution, while assessment of the genuineness of the applications deployed on the blockchain remains an open problem.

4 Architecture and design of Botract

In this section, we present the architecture and design of Botract. We initially explain the design concept and then move on to the construction of the smart contract used in the botnet application. Then, we demonstrate how to implement contract ownership for a bot commander, how commands are sent and received, and how bots are initialised for such communication. Finally, we characterise the entities involved and then describe the dataflow of botnet communication in Botract.

4.1 An overview of the design concept

Previously, researchers have described how to use the blockchain as a covert channel for botnet C2 by exploiting unused bytes in an outgoing transaction, such as unspendable output fields (Ali et al., 2017). However, instead of confining bot communication to a limited number of bytes, our approach exploits the inherent functionality of the smart contracts that help to enable blockchain-powered botnets. In our approach, bot commanders can develop their own C2 logic in the form of smart contracts (for example, using a language like Solidity, http://solidity.readthedocs.io/) that are deployed on the blockchain, and then exercise the functions of smart contracts by sending and receiving commands as well as keeping track of the state of bots. This is feasible because a blockchain like Ethereum is designed to be an open platform that allows third-party programs (i.e., smart contracts) to be deployed on the blockchain without being audited by an authoritative party.

4.2 Construction of the contract

Creating a smart contract on the blockchain is straightforward, and here, we demonstrate an example of a C2 contract using Solidity on top of Ethereum. However, the concept behind Botract is not restricted to Ethereum and can be implemented using other scripting languages and blockchain technologies that may be developed in the future.

4.2.1 Contract data structure

A smart contract can be loosely defined as an object-oriented class that consists of code (its functions) and data (its state) that reside at a specific address on the blockchain. In other words, a smart contract is an instance of a class that lives on the blockchain, and users can make calls to that instance through transactions.

The contract BotnetCnC shown in Figure 1 is an example of a smart contract that implements a simple botnet C2. As shown, a state variable for bot commands is declared as a data structure command. Members of this data structure include the command parameter cmd and the host address of the victim target. The cmd parameter specifies the current action that the bot must execute on behalf of the commander, and this command may vary from a simple ping request to a host scan or payload execution. The
**target** parameter specifies the host address that bots will attack, and this can be assigned as either a unicast host address, a list of addresses, or a specified range of addresses. Furthermore, this data structure can be modified to support command switches with additional arguments.

**Figure 1** Data structure of Botract’s smart contract (see online version for colours)

```solidity
contract BotnetCcC {  
  struct command {  
    string cmd;  
    string target;  
  }  
  address commander;  
  address contractAddress;  
  // unique requests  
  mapping(address => command) bots;  
  // broadcast requests  
  command broadcastToBots;  
  [..]  
}
```

The `commander` variable declares a state variable of type `address` suitable for storing the address of the bot commander. The `address` datatype is a 160-bit value that does not allow arithmetic operations and is used to reference an entity on the blockchain. The contract implements access control by checking the address of the contract owner in order to ensure that only the bot commander is allowed to change the state of the contract. These changes include modifying the list of commands to be sent to the bots, as well as the owner’s address and other variables.

Another address field that is stored within the contract body is the address of the contract itself `contractAddress`. The bot queries the contract for this address to determine if it needs to migrate to a new contract address. The commander may decide to migrate to a new address for various reasons such as fixing bugs or implementing a new functionality for the contract. This feature allows the commander to create an updated version of the contract and use it to redirect the bots to it.

The contract also includes a command field `broadcastToBots` to specify a broadcast command for the bots. The bots will query the field to grab the command(s) issued to them, and will parse and execute the command(s) stored in the field. Also, the contract may include a command table for unicast request for the bots `mapping (address -> command) bots`, essentially a mapping between the bot address and a command struct. In a similar manner, each bot will query the field of the command indexed by its address field.
4.3 Contract ownership

As mentioned previously, we use a smart contract to implement our C2 logic and facilitate communication between the commander and its bots. One key feature to implement for a smart contract that is not built in by default is contract ownership. Contracts in Ethereum are immortal by default – that is, they have no owners, and once they are deployed, their authors have no special privileges or control over them. Ownership is important especially for the bot commander because she needs to send commands to the bots using the contract, and this action can only be executed by the commander. In addition to integrity protection, the commander may also need to ensure that the privileges of reading the state of the contract are restricted to members of the botnet and that bots only have read access to the command fields of the contract. Furthermore, for obfuscation reasons, the commander may also need to migrate to a new contract address and to inform her bots about it to ensure seamless transfer. This means that the commander may need to destroy an old contract (programmatically) and create a new one during this migration process.

Figure 2  Contract constructor and ownership (see online version for colours)

```solidity
function BotnetCnC(address _b) {
    commander = _b;
    contractAddress = this;
}

function setContractAddress(address _c) {
    if (msg.sender == commander) {
        contractAddress = _c;
    }
}
```

For all of the above reasons, we need to explicitly define a static variable commander for the contract owner and assign the commander’s address to this variable when we create and initialise the contract and implement contract ownership. The code snippet above (Figure 2) depicts an example of ownership assignment for the `BotnetCnC` contract. As can be seen, a special function `BotnetCnC(address _c)` defines the constructor, which runs only once during the creation of the contract. A static variable `commander` of member type `address` is initialised with the constructor’s parameter that holds the bot commander’s address. Also, the contract address is initialised in the constructor by referencing the active contract using the assignment `contractAddress = this`. However, whenever the bot commander decides to deprecate the existing contract so that it is replaced with a new one, she may choose to set this variable to the new address so that bots can pick it up and migrate to the new contract.

The member `msg.sender` identifies the address of the caller, i.e., the address of the entity who initiated the function call, and this address can be used within the contract to validate ownership.
4.4 Sending bot commands

Sending commands to the bots is made easy by calling a setter function `setCommand()` in the contract. This function will simply update the `command` field with the requested command, the arguments, and the target address to attack. As shown in Figure 3, after the commander’s address is verified, the arguments of the commands are set to a temporary `command` struct before being recorded in the command field `broadcastToBots`.

Note that the way we assign the command structs `command(_c, _t)` to storage variables is to create a temporary variable in memory and then assign the function argument to it as a temporary placeholder before the assignment to the storage variable, `broadcastToBots`. The function arguments can also be assigned directly to the storage command field `broadcastToBots` because the Solidity compiler will automatically generate code that copies the struct data from memory into storage. A common mistake developers make is to declare a local variable and assign a memory object to it, which will not work in Solidity because the language defines local variables only as references to storage locations.

Figure 3  Sending bot commands via `setCommand()` method (see online version for colours)

```solidity
function setCommand(string _c, string _t)
    external returns (bool){
        if (msg.sender == commander){
            command memory temp = command(_c, _t)
            broadcastToBots = temp;
            return true;
        }
        return false;
    }
```

The `setCommand()`, depicted in Figure 3, is a function that sets commands as a broadcast request to all bots, but in order to implement unicast requests, we need to use a command table `(address → command) bots;` indexed by the bot address. This table creates a datatype that maps addresses to struct commands and can be viewed as a hash table. Therefore, in order to keep track of the bots, the commander needs to keep a list of all of the bots under her control. We also need to support a modified version of the `setCommand()` that includes the bot address as an argument and to write a wrapper that initiates the `setCommand()` calls to send commands for a specific bot address.

4.5 Receiving bot commands

It is relatively easy for bots to retrieve commands from the contract. All they need to do is to call the getter function `getCommand()`, so we obtain the command field from the contract and then parse the function’s response to execute the command. When a bot calls the `getCommand()` function (Figure 4), the contract will return the fields of the command struct as strings to the caller. In the case of a unicast request, the source address of the calling bot is obtained from the `msg.sender`, which can be used to identify the caller. As mentioned previously, a timestamp can be appended as a field in the
command struct, and the timestamp(s) can be used to decide on the expiry of a record, or to schedule a launch time, or both.

**Figure 4** Receiving bot commands via `getCommand()` method (see online version for colours)

```javascript
function getCommand() constant external returns (string, string) {
  return (broadcastToBots.cmd, broadcastToBots.target);
}
```

**Figure 5** Architecture diagram for the operation of Botract (see online version for colours)

**Figure 6** Bot script to call `getCommand()` function of the smart contract (see online version for colours)

```javascript
const contract = web3.eth.contract(abi).at(contractAddress);
var command = contract.getCommand.call({from: web3.eth.accounts[BOT]});
console.log("Command: " + command);
```

4.6 *Botract entities*

In the following, we describe the entities (players) that interact with each other under Botract:
Commander: the commander is the controller of the botnet who is responsible for deploying the smart contract that implements the C2 logic on the blockchain. She issues commands to the bots by calling the functions defined in the contract and keeps track of the bots under her control.

Contract/blockchain: the smart contract consists of a set of functions and data that implement the C2 logic. This contract is deployed on the blockchain, a distributed platform managed by a global peer-to-peer (P2P) network that overlays the Internet. Both the commander and the bots are clients of the blockchain and they issue function calls to the contract on a consistent basis to facilitate botnet communication.

Bots: the bots are a group of hosts that are controlled by Botract. They execute commands received from the smart contract to launch attacks, typically in the form of denial-of-service, against targeted victims. They may also be involved in surveillanceware or spyware purposed to leak information about users or launch commands on their behalf. Each bot comes pre-configured with the contract address of the C2 and a set of keys (i.e., an account) to interact with the blockchain.

Targets: the messages sent by the commander might instruct an infected machine to launch a DDoS attack on a particular target. In this context, a target is an external entity that is a victim of attacks from the botnet. In other words, a target is any entity attacked by the bots controlled through Botract.

4.7 Bootstrapping the bots

Now, we describe the deployment of the contract and the bot bootstrap process to initialise botnet communication. The Ethereum contract defines a set of functions that the commander and bots call to interact with each other. In the following, we describe the entities (players) that interact with each other under Botract.

Initially, the bot commander deploys the smart contract on top of the Ethereum blockchain and consequently pays a transaction fee in the form of some gas amount. As discussed, in order to achieve ownership, the bot commander will create an instance of the contract by unlocking her account to fund the blockchain (specifically, to fund miners or transaction verifiers on the blockchain) and makes a transaction to deploy the contract. Upon successful deployment, the commander will receive a contract address from the blockchain.

Subsequently, the bot commander will build a bot program configured with the contract address and account keys to access the blockchain. She will utilise side channels methods (e.g., phishing) to infect hosts with the bot to propagate the malware.

When the hosts are infected, they may initiate a connection to the C2 contract by calling a function to register themselves with the commander. This registration is optional and not required for the operation of Botract but can be used to inform the commander about live bots and to provide statistics about the botnet’s size and propagation in the wild. However, since bot registration requires an update of the
state of the contract, such as with a list of bot identifiers, the bots will have to pay a small fee in gas amount for their commits. In principle, the commander may choose to fund each bot for the registration fees prior to deployment by provisioning the funds to their accounts in advance so that they can register to the contract.

Figure 7  Bot commander script to call setCommand() function of the smart contract (see online version for colours)

```javascript
const contract = web3.eth.contract(abi).at(contractAddress);
var command = contract.setCommand.sendTransaction(cmd, target, botAddress, {from: sender});
console.log("Command: " + command);
```

4.8 Botract communication flow

After our discussion on bootstrapping, we next talk about the dataflow of botnet communication between the entities of Botract. A diagram of the communication flow for Botract is depicted in Figure 8.

1 The bot commander calls the function setCommand() to instruct the bots, and this requires an update on the state of the contract. The format of the attack command consists of a tuple of the:
   a command type
   b arguments
   c target host address(es)
   d optionally, the address(es) of the bot(s) who will launch the commands.

2 The bots will regularly pull new commands using the getCommand() function call to execute from the smart contract and send their addresses as arguments to this function. As a result, the function returns a command string that is parsed and executed by the bots. Instead of initiating a sendTransaction() primitive to execute function calls on the contract (which requires some gas amount), the bots will make a call to the function using the .call() primitive, which obtains a contract object and then references that object to execute a function locally on the host.

3 When the bots successfully retrieve the command by calling the getCommand() function call on the contract, they will determine whether the command they have received has been executed previously. They do this by checking if the timestamp of the command is within a predetermined threshold difference.

4 Finally, the bots will initiate the command against a list of target victims. This list is uploaded to the contract by the commander. The bots will download the list via a function call and execute the command to launch the attack.
5 Evaluation

In this section, we present the evaluation of Bottract, which includes implementation and analysis. First, we describe the proof-of-concept implementation of Bottract, which demonstrates the realisation of the method. Next, we investigate the storage footprint of the bots communicating with the blockchain. Finally, we calculate the economic cost by the function calls of the smart contract, in terms of Ether and dollar amount.

5.1 Experimental setup

We implemented a proof-of-concept for Bottract using a private blockchain under our control to demonstrate the feasibility of our approach. The first step in this process was to create a custom genesis file that defines the initial funds for our provisioned accounts. Then we arbitrarily set the mining difficulty and other parameters. Since our development was in a testnet environment, the Ether associated with the blockchain did not reflect real currency. We provisioned two accounts in this blockchain: one for the bot commander, and the other for the bot that received the commands.

We ran the Geth protocol daemon on the blockchain using the commander’s account address to launch a mining worker instance. The mining worker was set to run as a daemon to fund the commander’s account and would also process transactions on the blockchain. Next, we opened a command shell and attached to it to unlock the commander’s account. Afterwards, we funded the bot’s account with a small Ether amount, which was achieved by making a transaction from the commander’s account. Then, we created `iptables` rules and a `socat` listener to enable remote Geth connection for the bot through some arbitrary port, e.g., 6666. Finally, we deployed the smart contract (`BotnetCnC`) on the blockchain.

The contract was developed using the Solidity language, as shown in the above code snippets. We implemented additional scripts, merely wrappers to compile, deploy, and execute function calls in the smart contract, as shown in Figures 6 and 7. As for the bot implementation, we wrote a wrapper script that grabs the command from the contract and spawns a child process to execute the parsed string result of the function call. The scripts were written in Javascript, which imported `web3, solc` (solidity), and `fs` packages and executed using `nodejs`. Once executed, the scripts connected using a remote Geth connection to the server that stored the blockchain. As for the hardware used in the experiments, we used a MacBook Pro laptop and an Ubuntu server in our setup.
5.2 Bot storage minimisation

As of today, in order to communicate with the Ethereum blockchain, the commander and bots have to download the entire Ethereum blockchain which accumulates to more than 140GB (using Geth fast sync) (https://etherscan.io/chart2/chaindatasizefast). This may be considered as a limitation to our approach because typically, bots need to be stealthy in order to become pervasive. Nevertheless, the Ethereum community is already working to reduce the overhead for storing the blockchain on the client through the Light Ethereum Client initiative. According to Ethereum’s official website (https://github.com/ethereum/wiki/wiki/Light-client-protocol), the Light Ethereum Client will only download the headers of the blockchain, as opposed to entire blocks of transactions, thereby reducing the client storage footprint tremendously to about 100 MB (as promised). Though this will serve our proposal, as of March 2019, the Light Ethereum Client is still under development.

Another alternative to reduce the client’s storage footprint is to configure the bots to communicate with a remote Geth server(s) (which can be thought of as a decoy) that downloads the blockchain on their behalf and acts as a proxy in the peer-to-peer communication with the blockchain. The bots connect to the Geth server to transact with the blockchain and retrieve state updates. This solution allows the bots to delegate their storage overhead to the server, thereby reducing the client footprint to a few kilobytes; however, this would eliminate some of the proposed advantages of having the bots connect directly with peers on the blockchain. Nevertheless, this limitation may impact the bot’s resiliency to acquire the state of the blockchain, but it does not impact the availability of the smart contract itself.

5.3 Economics of Botract

Now, we describe our analysis of the cost to deploy and operate Botract. This cost is paid in the form of transaction fees that the commander funds in Ether. The deployment overhead includes the cost associated with uploading the C2 smart contract to the blockchain, and if the commander requires tracking of the bots, then additional funds are needed to register them and to record the identifier for each bot in the storage fields of the contract.

The deployment cost for the contract is calculated based on the number of low-level operations used, and this number varies depending on the contract size, the number of functions, and their complexity. The steps to calculate the contract deployment cost are straightforward. First, we disassemble the code for the smart contract, which is written in a high-level language like Solidity, into low-level operations that are run by the Ethereum virtual machine. Next, we estimate the number of function points for these opcodes in terms of gas. In general, hash computations and memory operations would cost more gas than arithmetic operations. In the third step, we determine the market rate for the gas price, which is based on the service level chosen to execute the operations.

To analyse our C2 contract example, we used the online tool Remix (https://remix.ethereum.org) and cross-validated with the documentation of the Ethereum API package provided for Geth and the web3 package. Based on this, we disassembled our Solidity code and calculated the total gas amount. As depicted in Table 1, the creation (deployment) of the contract code amounted to 4,712,388 gas. The transaction fees were calculated simply by multiplying the gas amount by the gas price. As
explained earlier, the value of the gas price is determined by the rate chosen. According
to the latest estimations by Gas Station (https://ethgasstation.info) (as of March 2019),
the cost for gas in the Ethereum blockchain is 2 Gwei for the the safe low rate, 2.5
Gwei for the standard rate, and 4 Gwei for the fast rate.

In our analysis, we computed the deployment cost on the real Ethereum blockchain
by taking the product of the latest currency price and the gas amount of the contract
deployed in our experiments. It turns out that the cost in US dollars to deploy
our contract can range from 1 to 3 dollars, depending on the mining service rates.
Furthermore, we also note that function calls to getCommand() that are invoked by the
bots do not add up to the total cost, since they do not consume any hash computations
or store operations. Interestingly, we note that a user can deploy and operate Botract
for less than 10 dollars if she decides to choose the safe-low rate. With this rate, the
commander can obtain at least 1000 calls at her disposal to send commands to the bots.

Table 1 Cost analysis for Botract function calls (March 2019)

<table>
<thead>
<tr>
<th>Blockchain</th>
<th>Gas costa</th>
<th>DeployContract()</th>
<th>SetCommand()</th>
<th>GetCommand()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>4,712,388 gas/call</td>
<td>25,465 gas/call</td>
<td>0 gas/call</td>
</tr>
<tr>
<td>Private chain</td>
<td>18 Gwei</td>
<td>0.00848 ETH</td>
<td>1.145925 ETH</td>
<td>0</td>
</tr>
<tr>
<td>Ethereum chain</td>
<td>Safe low (&lt;30 m)</td>
<td>0.00942 ETH ($1.32b)</td>
<td>0.0509 ETH ($7.13)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2 Gwei</td>
<td>0.0117 ETH ($1.65)</td>
<td>0.0636 ETH ($8.91)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Standard (&lt;5 m)</td>
<td>0.0188 ETH ($2.63)</td>
<td>0.10186 ETH ($14.26)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2.5 Gwei</td>
<td>0.0117 ETH ($1.65)</td>
<td>0.0636 ETH ($8.91)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Fast (&lt;2 m)</td>
<td>0.0188 ETH ($2.63)</td>
<td>0.10186 ETH ($14.26)</td>
<td>0</td>
</tr>
</tbody>
</table>

Notes: aEther from the private blockchain did not reflect real currency.
bCost estimations are based on the conversion rate of 1 ETH = $140.

6 Conclusions

In this paper, we have presented Botract, a messaging platform for botnet C2 that
exploits smart contracts and the blockchain. This inherently brings many advantages for
malicious authors not directly available with other approaches, such as programmability,
persistence, and resiliency. Most notably, our approach fundamentally exploits design
flaws in the Ethereum blockchain attributed to implicit end-user trust and the lack of
code scrutiny on publicly deployed contracts. As a result of this lack of governance,
anonymous users can deploy malicious code on the blockchain very easily without much
cost.

Despite the few challenges in implementing stealthy bots that interact with the
blockchain and the need for an initial monetary fund to deploy and operate these
botnets, this approach may be engineered to become effective for vast deployment in
the near future. Consequently, the blockchain community will need to come up with
tactical defenses to be implemented in the short term, without resorting to expensive
operations like hard forks of the blockchain codebase, and to fundamentally rethink the design of the blockchain with a security built-in mindset in the long-term. That being said, the development of a governance framework for auditing and deploying smart contracts is crucial to the flourishing of the blockchain technology itself. This is especially true for the new enhancements to the platform coming our way, such as Swarms (decentralised storage) and Whisper (decentralised messaging) frameworks (Felföldi, 2017), which are already being developed. Without proper built-in security in their design and architecture, these features (or extensions) of the blockchain technology may be abused in the future to a large extent.

Acknowledgements

This work was supported and funded by Kuwait University, Research Grant No. RX02/18.

References


