SANS Institute
Information Security Reading Room

Botnet Resiliency via Private Blockchains

Jonny Sweeny

Botnet Resiliency via Private Blockchains

GIAC RES 5500 Research Paper

Author: Jonathan Sweeny, j_sweeny@mastersprogram.sans.edu
Advisor: Johannes Ullrich

Accepted: 7 September 2017

Abstract

Criminals operating botnets are persistently in an arms race with network security engineers and law enforcement agencies to make botnets more resilient. Innovative features constantly increase the resiliency of botnets but cannot mitigate all the weaknesses exploited by researchers. Blockchain technology includes features which could improve the resiliency of botnet communications. A trusted, distributed, resilient, fully-functioning command and control communication channel can be achieved using the combined features of private blockchains and smart contracts.

1. Introduction

The connected, but distributed nature of botnets differentiates them from other types of malware. The person controlling the botnet, the bot herder, can manage it over a variety of communication channels, an activity known as command and control (C&C). This paper focuses on the resiliency of those communications channels – starting with a brief history of their evolution, then concentrating on the testing of a novel communications method: the private blockchain. Lastly, weaknesses in the model are acknowledged.

Terms that may be unfamiliar to readers are italicized when used the first time. These are defined in Appendix A.

1.1. Existing Bot Communication Channels

Historically, a bot herders' goal has been to make it difficult for network defenders to prevent and detect the activities of their botnets. In 1999, botnets utilizing Internet Relay Chat (IRC) for command and control emerged. These IRC-based botnets were efficient to operate and run for two reasons: they were versatile, and there was a large knowledge base among the bot herders, allowing them to reuse one another's code (Grizzard, Sharma, Nunnery, Kang, & Dagon, 2007). In the arms race since that time criminals have migrated to more resilient botnets. In 2007, criminals started using peer-to-peer (P2P) botnets because of improved resiliency over traditional C&C infrastructures, where a client/server model contained single points of failure (Wang, Wu, Aslam, & Zou, 2009).

In 2009, Jose Nazario from Arbor Networks reported that botnets started using Twitter for command and control. In that case, the tweets specified Base64-encoded links for the bots to visit. Nazario soon detected the behavior on the social networking sites, Jaiku and Tumblr, as well (Nazario, 2009). This use of popular internet services made it difficult for network security analysts to detect the command and control activities because they blended in with other internet traffic, known as "hiding in plain sight." A 2017 example of this technique was the use of visitor comments on Britney Spears' Instagram photos to control a botnet (Cimpanu, 2017). Specifically, the location of the C&C server was hidden in certain types of comments using hidden (non-printable) Unicode characters (Boutin, 2017). These examples illustrate the shift away from servers

that can be easily taken offline by law enforcement, but onto services that can be censored by the service owner's incident response team.

One technique to attempt to avoid detection is to communicate with the bots in an uncommon way, such as hiding the commands in ICMP packets (the protocol used by ping) instead of using the more-common TCP or UDP protocols. This specific method dates back to at least 1996, from The Loki Project (Phrack Magazine, vol. 7, issue 49, file 6). Taking that technique further, the Morto malware in 2011 sent commands to its bots via DNS TXT records, sent in response to DNS queries (Sancho, 2015). Morto was still vulnerable to takedown of the DNS server or seizure of the domain name. One method that protects domain names from seizure is the hidden service feature of the Tor anonymous proxy which protects the domains with private keys. In 2010, Dennis Brown of Tenable Network Security discussed the possibility of botnet command and control utilizing Tor hidden services (Brown, 2010; Casenove & Miraglia, 2014). These techniques can add resiliency to botnets but also increase their complexity.

In part to address that complexity, there has been a shift in malware over the last decade, much like the organized crime model where participants specialize in certain parts of the creation, distribution, management, and monetization of the product. Some individuals market their products or services in criminal marketplaces, a practice known as malware-as-a-service (Moreno, 2016). An example of this evolution is the TwitterNET Builder, a bot-creation kit which used Twitter accounts for C&C (Zeltser, 2015). This separation of duties allows individuals to specialize exclusively on botnet communication channels, focusing on improving resiliency.

In other efforts to avoid detection, researchers have demonstrated transmission of data covertly from air-gapped systems using the following media types: acoustic (frequencies beyond the range of human hearing), optical, seismic, magnetic, and thermal. Much of the research in these areas has been done by Dr. Mordechai Guri and his team at the Ben-Gurion University Cyber Security Research Center. While these techniques are interesting to study because the media types are less-commonly monitored by network defenders, they are limited in that in most cases they require that another nearby system is under the control of the attacker.

Jonathan Sweeny, j_sweeny@mastersprogram.sans.edu

An example of malware which used an unmonitored communication channel was reported in 2017 by the Microsoft Malware Protection Center. The researchers determined that the PLATINUM file-transfer tool used Intel's Active Management Technology (AMT) Serial-over-LAN (SOL), a chipset feature, as a channel for communication. This communication was independent of the operating system so it could not be blocked or detected by the hosts' firewall or monitoring software but was vulnerable to network detection (Kaplan, Sellmer, & Lelli, 2017).

Botnets' existing communication channels have consistently had trouble remaining hidden from the view of researchers and immune from takedown by law enforcement. The distributed nature of blockchains, along with certain security and reliability features together present a possible solution to such challenges.

2. Public and Private Blockchains

A *blockchain* is "a digital ledger in which transactions made in bitcoin or another *cryptocurrency* are recorded chronologically and publicly" (Blockchain - English Oxford Dictionary, 2017). Each sequential *block* file making up the blockchain contains new confirmed transactions since the previous block. Blocks are sometimes likened to individual pages of a ledger.

The first popular blockchain software, bitcoin, was released as open-source software in 2009 by a person or group using the name Satoshi Nakamoto (Nakamoto, 2009). While centralized transactional blockchains existed before, bitcoin was the first decentralized trustful blockchain (Aru, 2016). The bitcoin blockchain is public, meaning that any person can view transactions and see balances of addresses (accounts). Private blockchains differ from public ones in that "access permissions are more tightly controlled, with rights to modify or even read the blockchain state restricted to a few users, while still maintaining many kinds of partial guarantees of authenticity and decentralization that blockchains provide" (Buterin, 2015). For a bot herder, this means that access to the C&C channel can be restricted, but the communication channel can still be widely distributed and resilient. The bot herder can have complete control over which devices participate on a private blockchain.

2.1. Existing Research of Blockchains in Command and Control

Existing research related to the use of blockchain in botnet C&C only looks at the use of the public bitcoin blockchain. The two research articles are "A Novel Approach for Computer Worm Control Using Decentralized Data Structures" (Roffel & Garrett, 2014), and "ZombieCoin: Powering Next-Generation Botnets with Bitcoin" (Ali, McCorry, Lee, & Hao, 2015). These articles highlight the benefits of the blockchain's resistance to censorship or takedown because of the distributed nature of the bitcoin blockchain. This saves the bot herder from the cost and risk of operating another type of command and control system (Ali, McCorry, Lee, & Hao, 2015). The first risk avoided is that of getting caught by researchers or investigators when failing to use proper operations security (or "opsec") while interacting with C&C infrastructure. The second risk avoided is the single points of failure of C&C servers, which are either rented or hijacked. Additional benefits of blockchain C&C identified by these two papers include: anonymity, built-in state synchronization, and difficulty in determining the size of the botnet (Ali, McCorry, Lee, & Hao, 2015).

Gaps in existing research include studying the use of private blockchains in botnet command and control. Specifically, the features of access control and smart contracts offered by private blockchains can improve the resiliency of the botnet's communication channel by proving an adaptable command structure which is resistant to infiltration from researchers and law enforcement.

3. Ethereum Characteristics

Ethereum can be analyzed as a blockchain implementation and a basis for furthering the research of blockchain-based botnet C&C. The following sections explore several features of Ethereum and explain how they can be utilized to improve the resiliency of botnet command and control.

3.1. Accounts and Addresses

The most common use of blockchains is to transfer digital currency from one *account* to another. In Ethereum, each account on the blockchain is represented by a 20-byte *address*. An address is most simply thought of as a public key for an account. In order to

send cryptocurrency from one person's account to another, a transaction specifies both addresses along with the amount of the transfer and is signed by the private key of the sender. This signature is verified by other nodes which then record the transaction to a block in the blockchain. This understanding of the relationship and purpose of accounts and addresses is prerequisite to understanding smart contracts and how to restrict access to a private blockchain.

3.2. Smart Contracts

Ethereum was designed as a *smart contract* platform. A smart contract is a method of using computer code to initiate and enforce cryptocurrency transactions. Ethereum's origin is linked to a critique made by Vitalik Buterin that bitcoin was very limited regarding smart contracts (Araoz, 2016). A smart contract is made up of code, functions, and variables, which are written to the blockchain to be stored. When a smart contract is written to the blockchain, it can be referenced by its address. A simple example of a contract is a scheduled transfer: a certain date acts as a triggering event that causes a transfer of digital currency from one address (even the contract itself) to another address. A more-complex contract could manage a botnet by storing instructions for the bots in variables. Smart contracts not only define instructions and consequences but also administer the follow-up transaction(s), avoiding conflicts between the parties of the contract (BlockGeeks, 2016).

While an account has a password-protected private key and an address, a contract account has no private key but has an address, code, and storage (KLmoney, 2017). Contracts are owned by accounts. In the case of a botnet C&C contract, the owner of the contract would be the only member of the private blockchain able to modify the contract, giving orders to the bots.

3.2.1. Compilation and Execution

Most Ethereum smart contracts are written in Solidity, an object-oriented programming language. An online real-time compiler, debugging, and testing environment called Remix[1] provides an easy way to start writing code in Solidity

[1] http://remix.ethereum.org

(Introduction to Smart Contracts, 2017). The contract code is compiled to bytecode before being stored on the blockchain. After the contract is stored on the blockchain, its functions are available to the members of the blockchain who know its address and structure.

3.2.2. Function Types

Contracts have two main function types: read-only and transactional. Read-only functions are denoted with the keyword "constant" and can only perform computations and return values. Transactional functions can potentially change contracts or move funds (Araoz, 2016). A less-common type of function is the "event" – an empty function which helps users notice activity within the contract (Ethereum Foundation, 2017). A function can be affected by a "modifier" which allows it to inherit attributes such as requiring that the address of the account calling the function (also called "message sender" and denoted as "msg.sender") must match the address of the account that created the contract, or contract *owner* (Ethereum Foundation, 2017).

Contract functions can be created with the keyword "payable" which enables that function to store *ether*, the digital currency unit of the Ethereum blockchain. An example of a payable contract is a digital vault where cryptocurrencies are deposited and are later dispersed due to a function call or triggered event.

3.3. Ethereum Virtual Machine

The main feature of Ethereum is the Ethereum Virtual Machine. Jeffrey Wilcke, co-founder of Ethereum, describes it as follows: "The Ethereum Virtual Machine (EVM) is a simple but powerful, *Turing complete* 256bit Virtual Machine that allows anyone to execute arbitrary EVM Byte Code. The EVM … plays a crucial role in the consensus engine of the Ethereum system [and] allows anyone to execute arbitrary code in a trust-less environment in which the outcome of an execution can be guaranteed and is fully deterministic" (Wilcke, 2016). The EVM runs on each *node* connected to the blockchain and reads and writes to the blockchain. It verifies signatures to authenticate transactions. The code stored inside smart contracts is executed inside the EVM. Since the code is untrusted, the EVM runs in a sandbox (Github Contributors, Notes on the EVM, 2016).

To run the compiled bytecode, the EVM uses the contract's *application binary interface* (ABI), which is a JSON-formatted definition of the structures and methods used to access the functions. (Eva, 2015). Any node executing the contract's functions must have a copy of the ABI as well as the address of the contract on the blockchain. Therefore, in the case of a botnet command and control contract, the bots need both a copy of the ABI and the address of the contract.

3.4. Ether

The name of the cryptocurrency mined and transferred on the Ethereum blockchain is *ether*. Other units of measurement include finney (1/1000 ether), szabo (one millionth of an ether) and wei (one quintillionth of an ether, the smallest measurement). As with bitcoin, transfers of ether from one address to another typically include a transaction fee to incentivize *mining* nodes to process the transfer. Mining is the use of distributed computers to perform mathematical calculations for the blockchain, including the verification of transactions.

3.5. Gas

Transactional functions in smart contracts require *gas* to be spent and then the mining of a block to be added to the blockchain before any changes take effect. The purpose of gas is to prevent a miscreant or a mistake from causing a denial of service to the nodes executing contracts in the EVM. Similar to fuel being used when a car travels each mile, Ethereum gas is consumed by instructions running in the virtual machine. The amount of gas used for a given operation remains the same but the price of gas can vary as competition for execution time on the blockchain or the price of ether (which is used to pay for gas) varies (Chow, 2016). The way gas prevents a denial of service is twofold: contracts are required to pay gas for the instructions executed even if contracts fail to complete execution, and contracts have a "gas limit" value so that they do not go on forever. If the contract fails to execute completely, any changes are lost but the fee is still paid (Chow, 2016).

3.6. Geth

Geth is the main command-line console used to interact with the Ethereum blockchain. It is written in Go (an open-source programming language created by three employees at Google, also called "golang") and is an abbreviation of "Go Ethereum". Geth can be used to view the blockchain, mine ether, transfer ether between accounts, and create contracts (Github Contributors, Geth, 2017).

4. Utilizing Private Blockchains for Botnet Command and Control

Initial attempts to test private blockchains as a botnet communications channel involved putting the instructions for the bot in the comments of transactions moving ether between accounts. Testing in a laboratory environment, however, showed that these comments were not easily viewable by the bots, and furthermore introduced a problem of the bots knowing which transaction comments to check. The bots would have to check every transaction for instructions, a problem that could be exploited by researchers as a denial of service attack on the botnet C&C channel. Subsequent testing made it clear that the use of smart contracts would be more efficient because the instructions for the bots could be stored by the bot herder in variables within the smart contract. The bots could easily query these variables via functions within those contracts, by using the fixed address of the contract on the blockchain.

4.1. Updating Existing Contracts

After being stored on the blockchain, a smart contract cannot be altered unless it was built with functions designed to make such modifications. This is because blocks in blockchains are permanently stored and unalterable. The bot herder cares about this because the contract needs to be updated to change the commands sent to the bots – such as when the botnet changes from being idle to starting a distributed denial of service attack. The following are two methods that could be used to modify a contract:

Method 1) The first method is to have a certain set of variables in the contract that are modified by functions. This is trivial but may limit the ability to add new variables and functions, making it difficult to update the capabilities of the botnet's communications

channel unless the bots are updated as well. Since the addition of new functionality to the botnet was not a research objective, this method was selected, as the bot herder still has the ability to modify the bots' instructions by changing the variables.

Method 2) An alternative means of modifying the contract is to make use of an updatable address parameter which points to the address on the blockchain of the current version of the contract. This method could use the CALLCODE or DELEGATECALL features of Solidity to pass off execution to the functions in the newer contracts, even allowing those to access the first function's variables (Hess, 2016) (eth, 2017). This method would interest a bot herder who would want to create new communication channels – if, for example, a botnet was split into two distinct smaller botnets.

4.2. Access Control

The bot herder would implement access control on the contract so that it is not modifiable by other persons, including security researchers who *reverse engineer* the bot. The typical access control for a smart contract allows only the contract *owner* to update it – though a contract can be configured to have several owners and several different levels of access. This functionality is typically written in Solidity as follows:

```
if (msg.sender != owner) return;
```

As noted previously, "msg.sender" is always the address of the source of the function call (or contract creation). This line of code exits the function being executed with no action if the account calling the function is not the owner of the contract.

5. Botnet Smart Contract

For this research, a fully-functioning contract for managing bots was created and stored on a private Ethereum blockchain. Note that this is straightforward to build using open-source software. The contract was built with the capability to modify variables by including functions to perform the modification, as noted above – otherwise, the contract's variables must be static. The access control features were tested and it was confirmed that only the owner (or another person in possession of contract creator's private key and password) can update the contract's variables.

The following is a list of the functions created in the botnet C&C contract:

- Query bot version number (check for update)
- Modify the bot version number
- Query for URL to download new binary
- Modify URL download location of new binary
- Query command to be run
- Modify the command that bots should execute

Examples of commands that the bots might execute include: participating in a distributed denial of service attack, running a PowerShell command, starting a meterpreter (Metasploit) reverse TCP shell, executing a local program, or killing the bot's executable. The Solidity code for this botnet C&C contract is included in Appendix D.

6. Bot Herder's Interaction with Contracts

The first step for the bot herder is to create the private blockchain. This is done using a *genesis file* which is used to form the first block of the blockchain. The genesis file has a few configurable settings such as the mining difficulty and the limit to the amount of gas that can be mined per block. The bot herder may want to use the "alloc" setting to initialize one or more wallets containing a certain amount of ether. If this step is not taken, then blocks would need to be mined in order to have ether available in an account for the creation or modification of a contract.

6.1. Storing the Contract in the Blockchain

Once the blockchain is created, the next step is to store the contract in the blockchain. This is done in the following four steps (each is demonstrated in detail in Appendix B):

1. Write and compile the Solidity code.
2. Unlock the account – for ether payment to be made.
3. Upload the ABI file and the compiled code using the `LoadScript` command in geth. The address of the contract is returned.
4. Mine one or more blocks to have the contract added to the blockchain.

Once these steps are completed, any node on the blockchain can reference the address of the contract to execute its functions – the query functions are available to all accounts, but the update functions are only accessible by the contract owner. If the bot herder wants to prohibit researchers from querying these variables, then the functions can be restricted either by limiting membership to the private blockchain or by controlling access to the functions by only allowing certain addresses to run the queries.

6.2. Protecting Private Keys

Whenever the bot herder wants to update the contract's variables, the private key of the contract owner is needed. The keys are kept in the "keystore" folder of the geth client. The bot herder needs to be careful to protect the private key, as any keys left sitting on systems are vulnerable to acquisition by security researchers or law enforcement personnel.

7. Node Connectivity

Like other peer-to-peer networks, blockchains do not have central servers. They can be distributed across the globe, with each node attached to one or many others by a network connection. Some nodes may simply query the blockchain while others may store the entire blockchain, verify signatures, and field requests to confirm or validate transactions.

In order for a node to connect to others on the same blockchain, the following three things must be in place: they must each have the same genesis file, the geth clients must be launched with the same "networkid" value, and they must have a network path to one another. Nodes are connected to *peers* using geth's "admin.addPeer" command which requires the "enode" value (public key) of the node being added along with its IP address and port number. This process is demonstrated in detail in Appendix B.

Note that the traffic between nodes is signed by the node's private key and is verified by the public key (the "enode" value), making it hard to spoof communication from another node.

A bot herder will want to add several peers to each node to maintain resiliency in case one or more peers go offline. The ideal design would be a self-healing network. As nodes drop off, the bots could use a function in the contract to query for new nodes in order to establish replacement peers. These peer connections would ideally be rotated rather than static to mitigate the risk of takedown. If a central node was connected to all others, it would be easy for security researchers to enumerate the entire size of, or decapitate, the botnet by taking over that node. Viewing the list of peers is done using geth's "admin.peers" command.

8. Mining Speed

If the bot herder wants the bots to quickly respond to commands and cannot wait ten or more minutes for the new directives to take effect, then the speed of mining is important. This is especially relevant considering bitcoin had to make a soft fork in mid-2017 in order to improve transaction speed – a problem that arose from its popularity. Too many transactions were unable to fit into the limited size of each block (called the "block size limit") (Li, 2017).

Bitcoin is designed to produce a new block approximately every ten minutes. Ethereum, on the other hand, is designed to have a new block mined every twelve seconds (Github Contributors, Mining, 2017). This frequency means that a botnet using Ethereum for command and control would be able to communicate with bots quicker than a botnet using bitcoin. To further improve mining speed on a private Ethereum blockchain, the mining difficulty can be modified in the source code of geth before it is compiled. In a laboratory environment testing this modification, blocks were consistently mined in less than one second. Detailed instructions for this modification can be found in Appendix C.

9. Weaknesses & Countermeasures

The following are several vulnerabilities to using a private blockchain for the communication channel of the command and control of a botnet.

9.1. Joining the Private Blockchain

The first weakness is that it might be easy for researchers to join the private blockchain based on the information present in the bot software that could be identified via reverse engineering. Specifically, the researcher could extract the data for the genesis file, the network ID, and the "enode" identifier of a peer node. Using these three items, the researcher could use geth to connect to other nodes on the private blockchain. The researcher could then repetitively query the contract's variables to detect botnet behavior and track any changes. The bot herder might try to mitigate this by limiting or vetting each node that connects to the blockchain. The bot herder might also try to obfuscate or encrypt the names or values of the contract's variables, but the researchers presumably have the bot's code that decodes them, so these measures could also be reverse engineered.

If researchers took over any nodes and viewed the blockchain, they would be able to identify the IP addresses of those nodes' peers (geth command: "admin.peers"). This would help to not only estimate the size of the botnet, but also would provide a list of bots to report to Internet service providers or law enforcement for cleanup and remediation. This can be mitigated by rotating the central nodes to lessen the chance of one being taken over by a security researcher. The scope of this problem greatly depends on the connectedness of the nodes. The botnet would be both more resilient and less visible to researchers if the nodes were connected in a random network following a Gaussian distribution over the set of all nodes instead of using a hub-and-spoke design (Tyler, Asselbergs, Williams, & Moore, 2009 February; 31(2)).

9.2. Points of Failure

One single point of failure would be the loss of the contract owner's private key. If the bot herder lost it, he or she would not be able to update the contract to modify the instructions to the bots. A related weakness is that the contract owner's private key and password might fall into the hands of researchers or law enforcement. If this happened, they would be able to control the botnet and could dismantle it. This can be mitigated by being careful where the key file is copied and by adding another layer of public/private keys to the instructions. For example, the bot herder signs or encrypts all instructions

with a private PGP key that is never stored on a blockchain node. This extra step, however, would add complexity to the bots' endpoint software in order to verify signatures. Even if researchers did not have the second private key, they could still DDoS the botnet by storing gibberish in the contract's variables if they only obtained the contract owner's private key and password.

Another possible point of failure would be loss of some of the blockchain's central nodes, either by being powered down or disinfected, which could result in some nodes becoming headless and disconnected from the rest of the blockchain. Security researchers could attempt to orchestrate this deliberately by flooding the botnet with many nodes under their control. Some of their nodes would eventually become well-connected and then if the researchers shut them all down at the same time, other infected nodes could become headless.

9.3. Anonymity

If the bot herder does not use anonymizing networks (such as Tor or VPN) when updating the contract's variables, their true IP address is viewable by at least one or more peers on the blockchain. Correlation of the timing of the new command with the node/peer coming online could result in the identification of the bot herder's IP address.

9.4. Network Detection

Like most botnet communication channels, the private blockchain is susceptible to network detection. If the bot herder does not change geth's default port from 30303 to something more common like 80 (using the parameter "--port"), then the activity is easily detected on networks that do not typically have Ethereum clients. Network engineers may block this port or use signatures from an intrusion detection system (such as Bro, Snort, or Suricata) to identify use of geth or Ethereum's communication channels.

A challenge for network security engineers is that some organizations might be unable to outright block blockchain communication such as bitcoin and Ethereum because there may be legitimate use on their networks. In these organizations, network engineers won't be able to block the protocols entirely and will instead have to create and implement signatures to match and block each strain of malware.

Jonathan Sweeny, j_sweeny@mastersprogram.sans.edu

9.5. Endpoint Monitoring

Endpoint security tools could be configured to block node-to-node communications, execution of unfamiliar code (such as geth), or other characteristics of the botnet. Examples of this technique are the monitoring for the MD5 hash of the executable, or the software fingerprints of the geth client or the bot code. Private blockchains do little to mitigate this area of risk that exists with most malware variants. Some malware is memory-resident (not written to the hard drive) to resist this vulnerability, but any code running on the endpoint is susceptible to detection by the host.

10. Further Research

Further research could be conducted to create a series of more-complex contracts that add new features to the command and control channels with each new version. Such research would require the use of pointers directing the bots to the address of the current contract. This could be necessary, for example, if a botnet grew large enough that the bot herder wanted to split it in half and rent each half out to different customers to be tasked to their individual needs. Another example where complex contract modification would be necessary, would be a change in operation of the botnet, such as installing ransomware on each bot. The malware could store the decryption keys in contract variables which can only be retrieved by the contract owner's private key. In this case, the bot herder may write a *decentralized application* whose functions are stored and executed on the blockchain but whose front end lives on a Tor hidden service.

A second area for possible research involves the area of network detection of the command and control activity generated by this botnet implementation. Research of the protocols with analysis of captured network packets could be used to create network signatures for tools like Bro, Snort, and Suricata to detect both the use of Ethereum and this command and control activity.

11. Conclusion

Many of the current successful botnets still use pseudo-randomly generated domain names for command and control (Bader, 2015) or hide communications in social network

profiles or comments (Cimpanu, 2017; FireEye, 2015). The use of private blockchains for command and control removes the following weaknesses that those current botnets face: it is easy to blacklist or whitelist domain names queried by a DNS server, pseudo-random domain names are typically easy to spot, domain names can be seized by law enforcement, and social network profiles and comments can be removed by the site owners.

The main improvement that blockchains offer is the resiliency from takedown that is gained from their distributed nature, and because the blocks of the chain are immune to censorship. Furthermore, private blockchains add access control features to the blockchain and its contracts. The Turing-completeness of the Ethereum Virtual Machine allows contracts to be written to meet the needs of any bot herder. The weaknesses in using blockchains for command and control of a botnet can be addressed by using Tor and an additional layer of private keys. Due to the success of laboratory testing, private blockchains utilizing smart contracts are definitely a viable option for a botnet command and control channel.

Jonathan Sweeny, j_sweeny@mastersprogram.sans.edu

References

Ali, S., McCorry, P., Lee, P. H.-J., & Hao, F. (2015). *ZombieCoin: Powering Next-Generation Botnets.* UK: Newcastle University.

Araoz, M. (2016, July 29). *The Hitchhiker's Guide to Smart Contracts in Ethereum*. Retrieved from Zeppelin Solutions: https://blog.zeppelin.solutions/the-hitchhikers-guide-to-smart-contracts-in-ethereum-848f08001f05

Aru, I. (2016, August 29). *The Chicken and Egg: Blockchain Existed Before Bitcoin*. Retrieved from The Cointelegraph: https://cointelegraph.com/news/the-chicken-and-egg-blockchain-existed-before-bitcoin

Bader, J. (2015, February 20). *The DGAs of Necurs*. Retrieved from Blog of Johannes bader: https://www.johannesbader.ch/2015/02/the-dgas-of-necurs/

Blockchain - English Oxford Dictionary. (2017). Retrieved from English Oxford Dictionary: https://en.oxforddictionaries.com/definition/blockchain

BlockGeeks. (2016, November 6). *Smart Contracts: The Blockchain Technology That Will Replace Lawyers*. Retrieved from BlockGeeks: https://blockgeeks.com/guides/smart-contracts/

Boutin, J.-I. (2017, June 6). *Turla's Watering Hole Campaign: An Updated Firefox Extension Abusing Instagram*. Retrieved from WeLiveSecurity: https://www.welivesecurity.com/2017/06/06/turlas-watering-hole-campaign-updated-firefox-extension-abusing-instagram/

Brown, D. (2010, August 1). *Resilient Botnet Command and Control with Tor*. Retrieved from Defcon: https://www.defcon.org/images/defcon-18/dc-18-presentations/D.Brown/DEFCON-18-Brown-TorCnC.pdf

Buterin, V. (2015, August 7). *On Public and Private Blockchains*. Retrieved from Ethereum Blog: https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/

Casenove, M., & Miraglia, A. (2014). *Botnet over Tor: The Illusion of Hiding* . Retrieved from NATO Cooperative Cyber Defence Centre of Excellence: https://ccdcoe.org/cycon/2014/proceedings/d3r2s3_casenove.pdf

Chow, J. (2016, June 23). *Ethereum, Gas, Fuel & Fees*. Retrieved from Consensys: https://media.consensys.net/ethereum-gas-fuel-and-fees-3333e17fe1dc

Cimpanu, C. (2017, June 6). *Russian State Hackers Use Britney Spears Instagram Posts to Control Malware*. Retrieved from BleepingComputer: https://www.bleepingcomputer.com/news/security/russian-state-hackers-use-britney-spears-instagram-posts-to-control-malware/

Dietrich, C., & Bureau, P.-M. (2015). *Hiding in Plain Sight: Advances in malware covert communication channels*. Retrieved from Black Hat 2015: https://www.blackhat.com/docs/eu-15/materials/eu-15-Bureau-Hiding-In-Plain-Sight-Advances-In-Malware-Covert-Communication-Channels-wp.pdf

eth. (2017, Jan 23). *Ethereum*. Retrieved from StackExchange: https://ethereum.stackexchange.com/posts/3672/revisions

Ethereum Foundation. (2017). *CREATE YOUR OWN CRYPTO-CURRENCY*. Retrieved from Ethereum Foundation: https://www.ethereum.org/token

Eva. (2015, October 28). *A 101 Noob Intro to Programming Smart Contracts on Ethereum*. Retrieved from Consensys: https://consensys.github.io/developers/articles/101-noob-intro/

FireEye. (2015, May 14). *Hiding in Plain Sight: FireEye and Microsoft Expose Chinese APT Group's Obfuscation Tactic*. Retrieved from FireEye Blog: https://www.fireeye.com/blog/threat-research/2015/05/hiding_in_plain_sigh.html

Foster, J. C. (2015, July 7). *The Rise of Social Media Botnets*. Retrieved from Dark Reading: http://www.darkreading.com/attacks-breaches/the-rise-of-social-media-botnets/a/d-id/1321177

Github Contributors. (2016, November 15). *Installation Instructions for Ubuntu*. Retrieved from Github: https://github.com/ethereum/go-ethereum/wiki/Installation-Instructions-for-Ubuntu

Github Contributors. (2016, July 21). *Notes on the EVM*. Retrieved from Github: https://github.com/CoinCulture/evm-tools/blob/master/analysis/guide.md

Github Contributors. (2017, August 1). *A Next-Generation Smart Contract and Decentralized Application Platform.* Retrieved from Github: https://github.com/ethereum/wiki/wiki/White-Paper

Github Contributors. (2017, June 10). *Ethereum Contract ABI*. Retrieved from Github: https://github.com/ethereum/wiki/wiki/Ethereum-Contract-ABI

Github Contributors. (2017, May 12). *Geth*. Retrieved from Github: https://github.com/ethereum/go-ethereum/wiki/geth

Github Contributors. (2017, June 29). *Mining*. Retrieved from Github: https://github.com/ethereum/wiki/wiki/Mining

Grizzard, J. B., Sharma, V., Nunnery, C., Kang, B. B., & Dagon, D. (2007, April 3). *Peer-to-Peer Botnets: Overview and Case Study*. Retrieved from Usenix: https://www.usenix.org/legacy/event/hotbots07/tech/full_papers/grizzard/grizzard_html/index.html

Hess, T. (2016, July 12). *Ethereum*. Retrieved from StackExchange: https://ethereum.stackexchange.com/posts/190/revisions

Introduction to Smart Contracts. (2017). Retrieved from Solidity: http://solidity.readthedocs.io/en/develop/introduction-to-smart-contracts.html

Kaplan, D., Sellmer, S., & Lelli, A. (2017, June 7). *PLATINUM continues to evolve, find ways to maintain invisibility*. Retrieved from https://blogs.technet.microsoft.com/mmpc/2017/06/07/platinum-continues-to-evolve-find-ways-to-maintain-invisibility/

KLmoney. (2017). *Chapter 2: Working With Contract Wallets*. Retrieved from KLmoney: https://klmoney.wordpress.com/beta7-contract-wallets/

Li, M. Y. (2017, June 16). *August 1st And The End Of Bitcoin?* Retrieved from Seeking Alpha: https://seekingalpha.com/article/4081991-august-1st-end-bitcoin

Moreno, M. (2016, March 31). *Malware as a Service: As Easy As It Gets*. Retrieved from Webroot: https://www.webroot.com/blog/2016/03/31/malware-service-easy-gets/

Nakamoto, S. (2009, February 11). *Bitcoin open source implementation of P2P currency*. Retrieved from P2P foundation: http://p2pfoundation.ning.com/forum/topics/bitcoin-open-source

Nazario, J. (2009, August 13). *Twitter-based Botnet Command Channel*. Retrieved from Arbor Networks: https://www.arbornetworks.com/blog/asert/twitter-based-botnet-command-channel/

Roffel, D., & Garrett, C. (2014). *A Novel Approach For Computer Worm Control Using Decentralized Data Structures.* Santa Cruz: University of California.

Sancho, D. (2015, May 11). *Steganography and Malware: Concealing Code and C&C Traffic*. Retrieved from TrendMicro TrendLabs Security Intelligence Blog: http://blog.trendmicro.com/trendlabs-security-intelligence/steganography-and-malware-concealing-code-and-cc-traffic/

StackExchange Contributors. (2016, July 17). *Is it possible to change the block target time?* Retrieved from StackExcahange: Ethereum: https://ethereum.stackexchange.com/questions/7141/is-it-possible-to-change-the-block-target-time/7142

StackExchange Contributors. (2017, July 28). *Modify string in contract*. Retrieved from StackExchange: Ethereum: https://ethereum.stackexchange.com/questions/23191/modify-string-in-contract-error-invalid-address

Tyler, A. L., Asselbergs, F. W., Williams, S. M., & Moore, J. H. (2009 February; 31(2)). Shadows of complexity: what biological networks reveal aboutepistasis and pleiotropy. *Bioessays*, 220-227.

Wang, P., Wu, L., Aslam, B., & Zou, C. C. (2009, July 20). *A Systemic Study on Peer-to-Peer Botnets*. Retrieved from University of Central Florida: http://www.cs.ucf.edu/~czou/research/P2P-Botnet-ICCCN09.pdf

Wilcke, J. (2016, May 18). *Optimising the Ethereum Virtual Machine*. Retrieved from Medium: https://medium.com/@jeff.ethereum/optimising-the-ethereum-virtual-machine-58457e61ca15

Zeltser, L. (2015, February 14). *When Bots Use Social media for Command and Control*. Retrieved from https://zeltser.com/bots-command-and-control-via-social-media/

Appendix A: Terminology

- **Account** – represented by an address, an account on the blockchain may contain ether, and may own and manage contracts.
- **Address** – an alphanumeric string which corresponds to a user's public key, and is used as a source/destination in a transaction to send/receive cryptocurrency.
- **Application binary interface (ABI)** – The JSON-encoded information about the structure of the contract. The ABI describes how the bytecode is separated into functions and what types of input are expected to each.
- **Block** – a file recording confirmed transactions since the previous block; sometimes to a single page of a ledger.
- **Blockchain** – a set of sequential blocks, each containing verified transactions by a distributed network of computers.
- **Cryptocurrency** – a digital currency which uses cryptography to verify transactions and to be generated through mining.
- **Decentralized applications (DApps)** – applications whose backend lives on a blockchain or other decentralized infrastructure.
- **Double-Spending** – an attempt to trick a recipient by sending a digital currency more than once: the first time to the victim and then again to himself. If the transaction is not correctly validated, the victim may incorrectly think he's been paid.
- **Ether** – the digital cryptocurrency of the Ethereum blockchain.
- **Ethereum** – a blockchain technology designed to support smart contracts.
- **Ethereum virtual machine** – the sandboxed virtual machine each node of the blockchain runs, they execute contract bytecode and verify transactions and messages.
- **Gas** – a measure of the computational power needed to process a smart contract, gas is used to calculate how much ether must be paid to create or modify a contract.
- **Genesis file** – the configuration file to start a new Ethereum blockchain. Each node on a blockchain must have the same genesis file.

Jonathan Sweeny, j_sweeny@mastersprogram.sans.edu

- **Geth** – a console used to interact with an Ethereum blockchain, written in the Go programming language – got its name from "Go Ethereum".
- **Go** – programming language developed by a group of Google employees.
- **Mining** – the use of distributed computers to perform mathematical calculations for the blockchain, including the verification/confirmation of transactions. To incentivize mining, these systems' owners collect the transaction fees paid by the submitters of the transactions.
- **Node** – one of many network-connected devices which is participating in the same blockchain. Various blockchains implement nodes differently, but common features of nodes include verification of signatures, checking transaction format, and mitigating the risk of *double-spending*.
- **Owner** – the owner of a smart contract is the account address on the blockchain that created the contract.
- **Reverse engineer** – the process of analyzing computer code to determine its behavior and characteristics.
- **Smart Contract** – a set of code including functions and variables that are stored in the blockchain, a smart contract can enforce parameters and pay out cryptocurrency.
- **Solidity** – an object-oriented programming language, the most common one used to create contracts on the Ethereum blockchain.
- **Transaction** – an entry on the blockchain that represents either a transfer of ether from one account to another or the creation of a contract.
- **Turing complete** – the system can encode any computation that can be conceivably carried out, including infinite loops.

Sources: (Introduction to Smart Contracts, 2017; Github Contributors, Ethereum Contract ABI, 2017).

Appendix B: Creating Private Blockchains & Compiling Contracts

The following instructions are for creating a private blockchain on the Ubuntu operating system (Github Contributors, Installation Instructions for Ubuntu, 2016):

1. Install Geth:

```
sudo apt-get install software-properties-common
sudo add-apt-repository -y ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install ethereum
```

2. Create a directory to store the blockchain and private keys, such as /home/user/ethereum/
3. Create the genesis.json file in that folder; example at: https://medium.com/blockchain-education-network/use-geth-to-setup-your-own-private-ethereum-blockchain-86f1200e6d40
4. Initialize the node setup: `geth –datadir "/home/user/ethereum/node1" init "/home/user/ethereum/genesis.json"`
5. Launch the geth console using the newly-created node: `geth --datadir "/home/user/ethereum/node1" --nodiscover --networkid 112233 console 2>console.log`. The following are all commands to be run within the geth console:
6. View your node's enode value: `admin.nodeInfo`
7. To add a second node, repeat steps 4 & 5 but replace "node1" with another unique value and add a port setting on step 5 so there is not a conflict using the default port of 30303. Example: `--port 40404`.
8. To have the nodes connect to one another, use the "addPeer" command from the second node as follows: `admin.addPeer(enode)`, where *enode* is the enode value seen in step 6.
9. To confirm that the nodes are connected, run `admin.peers` from either (or both) nodes.
10. Before mining can begin, accounts need to be created: `personal.newAccount("password")`. The output of this command will show the

address of the account created. The password you select will be needed any time you spend ether from the account (including creating contracts).

11. Start mining by running `miner.start(4)` where 4 is the number of threads you want.
12. To view the last block mined: `eth.getBlock('latest').`
13. To spend ether, the account needs to be unlocked as follows: `personal.unlockAccount("0x3215…8a38", "password").` Alternatively the password ca be left off and will be requested via prompt. This is safer so it is not stored in the command history.
14. To view the balance of your profile's default account:

```
web3.fromWei(eth.getBalance(eth.coinbase), "ether").
```

Compiling a Contract

The following steps outline the process for compiling a contract, storing it to the blockchain, and then executing its functions (StackExchange Contributors, 2017). This example uses the Solidity programming language and the geth console.

1. Write the Solidity code for the contract. See sample code in Appendix D. In this example, the code is written to a file named `c2.sol`
2. Compile the Solidity code as follows: `solc -o . --bin --abi c2.sol`. The output of this function will be two files, both with the name of the contract ('c2' in this case) with the extensions of .bin and .abi. The c2.bin file contains the compiled bytecode and the c2.abi file contains the application binary interface information (the structure of the contract).
3. Edit c2.abi to add a variable name to the ABI code as follows: `var c2Contract = eth.contract(…)` where the "…" represents the previous file contents.
4. Edit c2.bin similarly as follows:

```
personal.unlockAccount(eth.accounts[0])

var c2 = c2Contract.new(
 { from: eth.accounts[0],
     data: "0x...",
     gas: 500000
```

```
        }
    )
```

5. Launch the geth console
6. Load the two files just edited as follows:

```
loadScript("c2.abi")

loadScript("c2.bin")
```

The second command will request the passphrase of the account used because it costs ether to write the contract to the blockchain. The output should show "Submitted contract creation" along with a timestamp and the address of the contract on the blockchain. The contract functions cannot be run, however, until the next block has been mined.

7. To mine the next block, type `miner.start(1)` in the geth console where 1 is the number of threads to mine with.
8. After one or more block has been mined, the functions within the contract can be executed. To list the available functions, you can just type the contract name in the geth console. To run a read-only function, just type it like this: `c2.queryLink().`
9. To run a transactional function (one which updates the variables), an account needs to be specified which will pay for the ether. This can be done in two different ways:
    a. In one line like this: `c2.updateV(4321,{from: eth.accounts[0]})`
    b. Or in two steps like this:

```
web3.eth.defaultAccount = eth.accounts[0]
c2.updateV(4321);
```

10. Do not forget to mine one or more blocks after updating values in order to have queries reflect the new setting.

Appendix C: Modifying Mining Difficulty

To improve the mining frequency, the mining difficulty is eliminated by modifying the geth client's source code. The following steps detail the procedure:

1. Clone the geth repository: `git clone https://github.com/ethereum/go-ethereum`
2. Install golang:

   sudo add-apt-repository ppa:gophers/archive

   ```
   sudo apt-get update
   sudo apt-get install -y build-essential golang
   ```

   a. If you get errors about the correct version of Go, manually installing this version helps: `wget https://storage.googleapis.com/golang/go1.7.1.linux-amd64.tar.gz`
3. Modify the mining difficulty by editing the following file in your local geth repository: `go-ethereum/consensus/ethhash/consensus.go`. Edit the `CalcDifficulty` function (line 289 as of 2017-08-05) by removing the entire body of the function and replacing it with:

   ```
   return big.NewInt(1)
   ```
4. Compile geth from source:

   ```
   cd go-ethereum
   make geth
   ```
5. Launch the compiled version of geth via `build/bin/geth`

Source: (StackExchange Contributors, Is it possible to change the block target time?, 2016)

Jonathan Sweeny, j_sweeny@mastersprogram.sans.edu

Appendix D: Smart Contract Command & Control Sample Code

This is a copy of the Solidity code developed by the author.

```
pragma solidity ^0.4.13;

contract c2 {
    address owner;
    uint currentVersion;
    string command; //valid values include: ddos {target}, PS {powershell_code},
kill, metrepreter {IP}, anything else is executed as is.
    string bot_url;
    string c2list;

    //Constructor is automatically executed upon creation:
    function c2(){
        owner = msg.sender;
        currentVersion = 17;
        command = "psexc.exe \\server c:\temp\svchost.exe";
        bot_url = " http://bit.ly/1sG87iJ";
        c2list =
"enode://7a1498b4e4dadd444c453c81e91ddce0140d3fee005ada8cc93501e166d8fda6c352fa
709763d44d9d464f1b1cfaf6ebbdd4e2fde573a56cb3a8706a791b8eb8@192.168.96.50:30303"
;
    }

    // Update the command for the bots to run:
    function updateC(string newish){
        if(msg.sender != owner) return;
        command = newish;
    }

    // Update the command for the bots to run:
    function updateL(string s){
        if(msg.sender != owner) return;
        bot_url = s;
    }

    // Update the command for the bots to run:
    function updateV(uint v){
        if(msg.sender != owner) return;
        currentVersion = v;
    }

    // Update the list of C2 nodes on our blockchain.
    function updateC2(string s){
        if(msg.sender != owner) return;
        c2list = s;
    }

    // Current command bots asked to execute:
    function queryC() constant returns (string){
        return command;
    }

    // URL to download current version of bot:
    function queryLink() constant returns (string){
        return bot_url;
    }
```

Jonathan Sweeny, j_sweeny@mastersprogram.sans.edu

```
    // Current version of bots code:
    function queryV() constant returns (uint){
        return currentVersion;
    }

    // Current list of c2 enodes; this is so the bots know which peers to add.
    function queryC2() constant returns (string){
        return c2list;
    }
}
```