Tearing up Smart Contract Botnets

Jonathan Sweeny

Tearing up Smart Contract Botnets

GCIA

Author: Jonathan Sweeny, j_sweeny@mastersprogram.sans.edu
Advisor: Johannes Ullrich
Accepted: 9 October 2018

Abstract

The distributed resiliency of smart contracts on private blockchains is enticing to bot herders as a method of maintaining a capable communications channel with the members of a botnet. This research explores the weaknesses that are inherent to this approach of botnet management. These weaknesses, when targeted properly by law enforcement or malware researchers, could limit the capabilities and effectiveness of the botnet. Depending on the weakness targeted, the results vary from partial takedown to total dismantlement of the botnet.

1. Introduction

A bot herder who manages a botnet is an activity known as command and control (C&C). Bot herders face many of the same challenges as a mafia boss who manages operations while incarcerated: they're both trying to direct criminal activity but want the instructions to reach the intended target without being intercepted, modified, or blocked. To mitigate these threats, bot herders are starting to use blockchains for a more resilient communications channel (Trend Micro, 2013). Private blockchains, in particular, offer valuable features to bot herders because the stored data can be regulated by the owner – for example, the C&C instruction data stored in the blocks can be blocked from public view (Sweeny, 2017). This paper focuses on the potential weaknesses of private blockchain-based C&C communication channels and explores specifically how malware researchers can take advantage of those weaknesses to dismantle the botnet.

1.1. Existing Research

In 2014, researchers began to speculate about botnets using Bitcoin's public blockchain as a communications channel (Roffel & Garrett, 2014). Private blockchains, however, are an enticing communications method for bot herders because the instructions to the bots can be access controlled but still incredibly resilient to censorship or alteration due to the blockchain. Only two scholars have published research about the use of *private* blockchains as a botnet C&C communications channel. The first was written by Jonathan Sweeny in 2017 ("Botnet Resiliency via private Blockchains"), and the second was written by Majid Malaika of omProtect ("Botract – Abusing Smart Contracts and Blockchain for Botnet Command and Control") at the SecTor conference (Malaika, Al Ibrahim, & Al Ibrahim, 2017). Both Sweeny and Malaika focused on Ethereum, the most popular private blockchain platform.

One of the central features of Ethereum is smart contracts, which is a feature that allows code and variables to be stored on the blockchain. The contract's variables can contain instructions for the bots such as *conduct a denial of service attack against 192.168.7.52* -or- *use this email template to send spam to these thousand email*

addresses. Unlike traditional blockchains, such as Bitcoin, the functions of the smart contract in Ethereum's private blockchains allow the variables to be both queried and updated. The Ethereum component that executes code is called the Ethereum virtual machine (EVM). The EVM, like most programming languages, is Turing complete – meaning that it can process any conceivable programming functionality. The EVM runs on each node of the private blockchain, and its purpose is to execute the functions of the contract, verifying the authenticity of transactions. Each node runs the same code, so the benefit of this distributed execution is the verification – not the efficiency.

Researchers show that using smart contracts stored in Ethereum private blockchains allow a robust and reliable method for a bot herder to control his or her botnet. The ability to update the contract's code and variables allow easy and uninterrupted management of the botnet's operations. These communication channels must be targeted by network defenders in order to dismantle the botnet (Sweeny, 2017).

1.2. Challenges for Network Defenders

Botnets have continuously been getting more difficult to dismantle due to the "arms race" between bot herders and researchers or network defenders. To dismantle a botnet in the past, law enforcement officials could simply ask an ISP to take a single server offline which was hosting all the botnet infrastructure. Recent botnets, however, have added resiliency through features such as C&C servers, fast-flux DNS, domain generation algorithms, peer-to-peer communication, SSL, steganography, anonymous networks, and the ability to hide C&C messages on social media platforms. This arms race has continuously created new challenges for researchers, and the use of private blockchains is no different.

When botnets use blockchains to issue direction, network defenders face the challenge that the blockchain is heavily connected and distributed and is also resilient against modification because of its use of private keys. Additionally, private blockchains control access to the blockchain and its contract's functions. These challenges introduce new obstacles for network defenders to overcome.

Author Name, email@address

1.3. Research Scope

This research focuses on the resiliency features of private blockchains' smart contracts as used as a command and control communication channel for botnets. Specifically, the focus will be on weaknesses associated with this method and how they might be targeted by researchers to disrupt or dismantle a botnet.

2. Weaknesses Targeted

The following sub-sections detail several opportunities for researchers to disrupt or dismantle a botnet which utilizes a private blockchain as its C&C communications channel. Each part includes a detailed explanation of the vulnerability along with specific steps to exploit it.

2.1. Enumerating and Fragmenting the Botnet

There are at least three ways to enumerate the membership and size of a botnet. The classic method is to monitor a C&C server's network traffic and to log the unique IP addresses that connect to it, seeking instructions. This technique becomes difficult, however, if the botnet is segmented or if the C&C servers change regularly. Two additional methods to enumerate the members of a private blockchain-based botnet would be to use the blockchain's built-in feature to enumerate nodes (the *admin.peers* function in the case of geth), or to monitor for Ethereum network traffic from a single member and to keep pivoting to identify additional nodes. This pivoting is done by looking at Ethereum's network traffic. Ethereum nodes use port 30303 (TCP or UDP) by default to communicate with one another.

No matter the method of enumerating the members of the botnet, the vulnerability to the botnet is a mass node takedown. Such a takedown can occur by removing large numbers of nodes – typically by notifying ISPs about infections and urging cleanup. It can also happen in a more targeted attack; this includes breaking the connections between nodes by targeting well-connected nodes, thereby fragmenting the botnet. Some node designs are more resilient to fragmentation – the more well-connected a node is, the less

likely it will be severed from the botnet. Node design, therefore, needs to consider that the more well-connected a botnet is, the easier it is for researchers to enumerate the nodes.

2.2. Tracking Bot Herder on the Network

At the heart of this vulnerability is the question: can the bot herder be identified by monitoring network traffic? This monitoring could take place at various points on the network such as the private blockchain, C&C server, or endpoints. The point of the monitoring would be to discover the true IP address and location of the bot herder. A careful bot herder will keep this in mind while creating the private blockchain and smart contract and when updating the contract's variables. To avoid disclosing his or her IP address and geographical location, the bot herder is likely to use anonymizing proxies and services like Tor.

2.2.1. Ethereum on the Network

As mentioned previously, Ethereum uses UDP & TCP ports 30303. It also uses a peer-to-peer protocol suite called RLPx (Github Contributors, 2018). To understand and identify weaknesses in botnets' use of Ethereum smart contracts, it helps to first know how Ethereum utilizes these protocols to transmit blockchain information.

In Ethereum's peer-to-peer (p2p) model, nodes send each other information about transactions and state when one notices it is missing information that a neighboring node possesses. The nodes compare the hash of their "best" block – the last valid block with the most work completed in that fork of the chain – and their chains of hashes to determine which blocks are needed (Github Contributors, 2018).

One of the protocols used to manage these p2p transactions is RLPx, a cryptographic network suite that provides node discovery, encrypted transport, flow control, peer reputation, and other security features (Github Contributors, 2018). The RLPx protocol ensures a well-connected network by establishing and maintaining connections to peer nodes (Gerring, 2016) (Jankov, 2018).

When nodes are connecting to one another, there are two phases: peer authentication, and base protocol handshake. During the first phase, the nodes establish an authenticated and encrypted communication stream. They also perform an encryption handshake to exchange ephemeral keys and set starting values for the encrypted session. During the second phase, the nodes cooperate to select protocols each supports. RLPx uses Keyed-Hash Message Authentication Code (HMAC FIPS 198) with SHA256 for authentication and AES-256 for encryption (Github Contributors, RLPx Encryption, 2018). The authentication and encryption of each frame individually (both headers and payload) provide integrity and confidentiality, severely impairing the ability of network defenders or researchers to modify or spoof traffic (Github Contributors, 2018).

2.2.2. Analysis of Ethereum's Network Activity

The popular *tcpdump* tool was used to collect network traffic from Ethereum nodes on a private blockchain while a contract was being uploaded, blocks were mined, and stored variables were queried. The script used to capture this traffic is provided in the Appendix. The packet capture in Figure 1 below shows many TCP packets using ports 30303 and 34034:

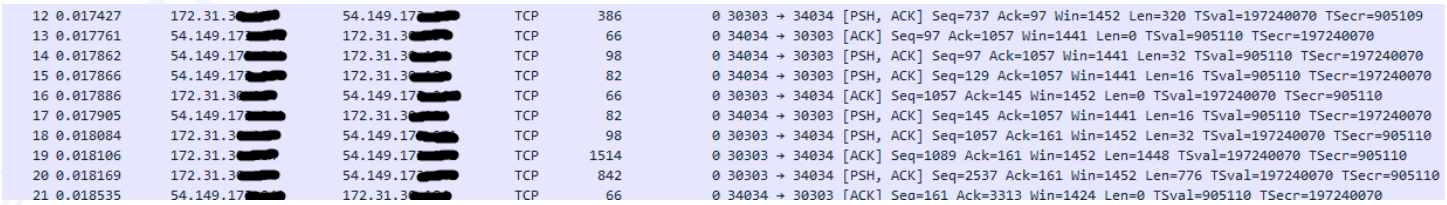

Figure 1. Network packet capture showing Ethereum packets between two nodes.

Because of the AES-256 encryption discussed above, the content of the packets is not decipherable. This is shown in Figure 2:

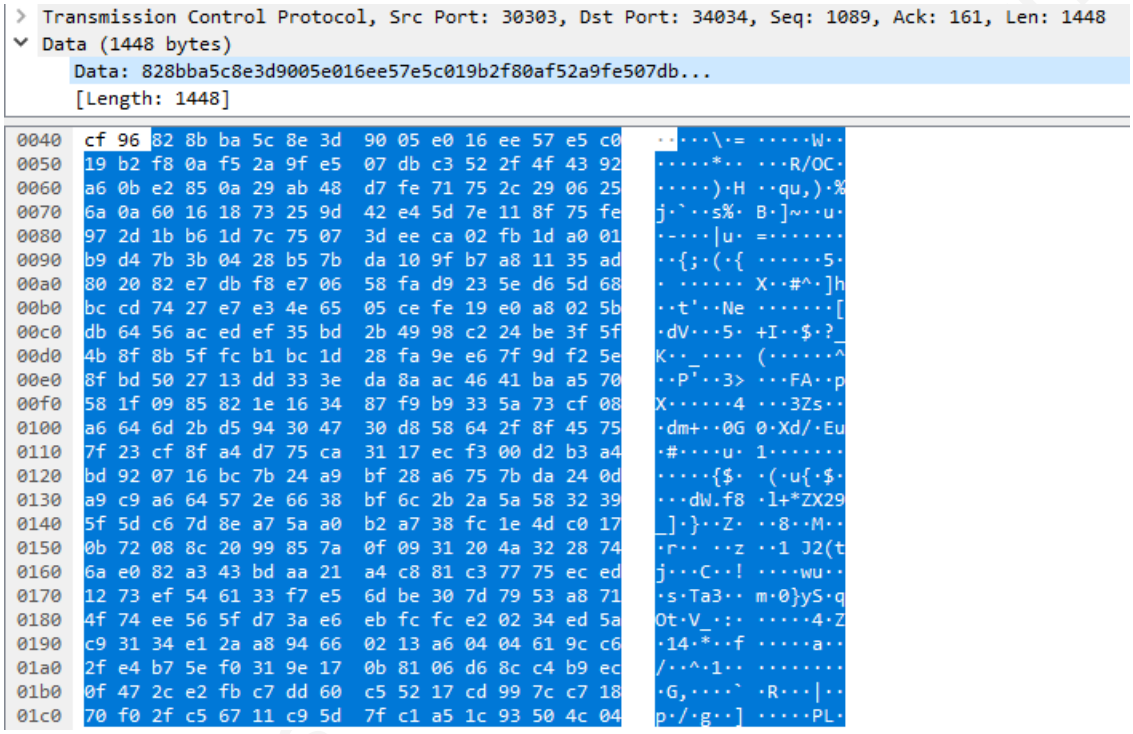

Figure 2. Content of a sample Ethereum packet between two nodes; note that the content in the right column is not decipherable text.

Raul Kripalani, a protocol engineer at PegaSys Engineering announced in June 2018 that his organization is developing and releasing an open-source Wireshark dissector for Ethereum's UDP p2p node discovery. Tools like this will help to parse (though not decrypt) Ethereum's network communications. He also announced future plans to work on the RLPx/TCP dissector (Kripalani, 2018) (ConsenSys, 2018).

Although intrusion detection tools like Snort cannot alert on the content of the encrypted traffic, Snort can attempt to detect the use of Ethereum. The following two sample Snort rules detect traffic using port 30303 (the default) via TCP and UDP:

alert tcp any 30303 <> any any (msg: "Ethereum_TCP"; sid:1030303; rev:1;)

alert udp any 30303 <> any any (msg: "Ethereum_UDP"; sid:1030304; rev:1;)

These rules are by no means specific enough to use in production without further testing and refining in a network defender's particular environment. These rules could match plenty of traffic that is not actually related to Ethereum. One simple improvement

would be to add a port number or range on the other side of the rule (replacing 'any' with '1025:', for example, to limit to ports 1025 and higher). This improvement would prevent false positive alerts from web browsing activity (where the client happens to use the ephemeral port 30303), for example.

Attempts to modify network traffic via a "man in the middle" (MITM) attack would face similar challenges due to the authentication and encryption of the Ethereum traffic. Proxy software such as Burp Suite which might modify traffic en route would cause the traffic to be dropped by the receiver because it would fail authentication. Therefore, the most effective network attack against Ethereum nodes at present is a denial of service attack. Such a response is most likely to take place on an intrusion prevention device or a network firewall by blocking the network traffic described above. This effort would prevent the bot from receiving new instructions and therefore most likely prevent it from participating with the botnet in any meaningful way.

2.3. Endpoint Detection

As with any malware running on an endpoint, detecting the client components of a private blockchain-based botnet would be successful via the traditional methods of looking at files, processes, and network sockets. Network defenders can detect botnet activity on computers by creating signatures to match these indicators or by detecting Ethereum clients where unexpected. This detection most often occurs via software like antivirus programs or by looking at the network activity from the endpoint.

2.3.1. Looking at the Geth Client

The most prevalent Ethereum clients are *geth* (written in Google's *Go* programming language) and Parity (written in Mozilla's *Rust* programming language) (Github Contributors, 2018). Geth stores data in the *leveldb* format, an open-source Google storage library based upon key-value pairs. Leveldb permits forward and backward transactions on data. Leveldb data is automatically compressed using Google's fast *Snappy* library (Saini, 2018). In both Linux and Windows, the geth client stores blockchain, transactional, and state data inside the *chaindata* folder. Figure 3 shows an

example of what the chaindata folder looks like – note the leveldb files by the .ldb extension:

```
ubuntu@Ether3:~$ ls -l c2/geth/chaindata/
total 28272
-rw-r--r-- 1 ubuntu ubuntu 2151475 Aug  8 02:19 000120.ldb
-rw-r--r-- 1 ubuntu ubuntu 2177196 Aug  8 02:19 000121.ldb
-rw-r--r-- 1 ubuntu ubuntu 2167455 Aug  8 02:19 000122.ldb
-rw-r--r-- 1 ubuntu ubuntu 2178941 Aug  8 02:19 000123.ldb
-rw-r--r-- 1 ubuntu ubuntu 2207766 Aug  8 02:19 000124.ldb
-rw-r--r-- 1 ubuntu ubuntu 2209733 Aug  8 02:19 000125.ldb
-rw-r--r-- 1 ubuntu ubuntu 2209708 Aug  8 02:19 000126.ldb
-rw-r--r-- 1 ubuntu ubuntu 2193580 Aug  8 02:19 000127.ldb
-rw-r--r-- 1 ubuntu ubuntu 2162867 Aug  8 02:19 000128.ldb
-rw-r--r-- 1 ubuntu ubuntu 2152977 Aug  8 02:19 000129.ldb
-rw-r--r-- 1 ubuntu ubuntu 2153009 Aug  8 02:19 000130.ldb
-rw-r--r-- 1 ubuntu ubuntu  447459 Aug  8 02:19 000131.ldb
-rw-r--r-- 1 ubuntu ubuntu 4487351 Aug 15 00:16 000132.ldb
-rw-rw-r-- 1 ubuntu ubuntu     183 Aug 18 23:46 000136.ldb
-rw-rw-r-- 1 ubuntu ubuntu       0 Aug 19 00:11 000167.log
-rw-rw-r-- 1 ubuntu ubuntu      16 Aug 19 00:11 CURRENT
-rw-r--r-- 1 ubuntu ubuntu       0 Jul 31 03:04 LOCK
-rw-rw-r-- 1 ubuntu ubuntu     178 Aug 19 00:11 LOG
-rw-rw-r-- 1 ubuntu ubuntu     178 Aug 19 00:07 LOG.old
-rw-rw-r-- 1 ubuntu ubuntu    1456 Aug 19 00:11 MANIFEST-000166
```

Figure 3. Screenshot of a *chaindata* folder from an Ethereum client on an Ubuntu computer.

The leveldb files in the above screenshot are easily queried with a short Python script (Github Contributors, 2016) as follows:

```
#!/usr/bin/python3
import plyvel
db = plyvel.DB('/home/ubuntu/c2/geth/chaindata/')
i=1
for key, value in db:
    print("\nkey {}: {}".format(i,key))
    print("value {}: {}".format(i,value))
    i+=1
```

Here is an example of the output of the script (one of the hundreds of key/value pairs):

```
key 1: b'\x00\x001\xb8\xa2\xcc\xd8\n\xadXZ\xc9\x07\x1e\x80C.\xab\x1c\x7f\xf5\rC\x90\x960\xff\t-
value 1: b'\xf9\x01q\x80\xa0\x1e\xac\x02\x92\xb7~\xe8\xb3~U;\x838\x95T\xae$\xa0\xd5\x835\x14\x9
\xcd\xa7\xf6\xa0\xec\x11\xa5\x1bR$\xf0IUk\x10!{\xe5\xf5|\xe2\x9b\xdlV#\x83\xfd\x80\xa0\xc9\xdf\
f\x95\xeb\xc6I\xb4\xaaL\xbcAf\x0e\xa0%B\x1d\xa17i\xbd\xc7\x9c\x86\x8e\x94*\x00Y\x9b\xcc\x9dJ\xf
2*x\xad\xa3\x00\x03\xc2\x06\xb5t\x1cU\x8a\x02\x810\xb4\xfd\xfd\xb7X\xae\t\x01\xa2\x8d\xa05e\x8e
94\xe6\x9e\xb9\x14Y\xc7\x84\x88\x0e\xd7\xc3\xb8 \x98g\xff\xbc\x8c\x9d\xa0\xf9Nn>aD\xc8\x0c\xf1g
```

Author Name, email@address

Figure 4. The key and value pairs from the leveldb files which store the Ethereum blockchain.

It is immediately obvious that the data is encoded – Ethereum uses a unique "Modified Merkle Patricia Trie" for data stored in leveldb (Saini, 2018) (Github Contributors, 2018) (Buterin, 2015). The tree (or "trie") is a popular storage format for storing sequences because it is compact. Ethereum, in fact, stores four different trees in the blockchain: state (account balances), storage (smart contract code), transaction, and receipt. This configuration is seen below in Figure 5:

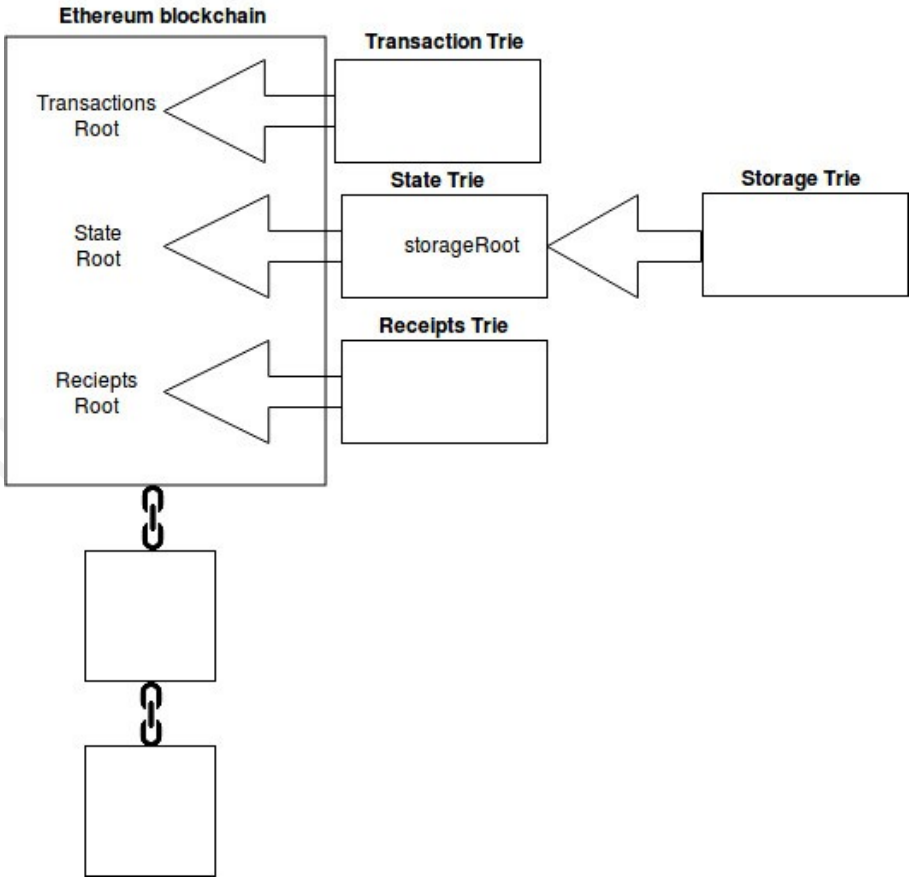

Figure 5. A summary of how Ethereum stores the several types of blockchain data. Image source: (Saini, 2018).

These separate trees allow permanent data like transactions to be stored separately from temporary data like account balances. This structure also permits *light* clients to generate easily confirmable answers to a variety of queries including (Buterin, 2015):

• Was transaction X included in block Y?

• List all instances of an occurrence (such as smart contract variable updates) in the past 90 days.

• What is my current account balance?

When attempting to modify these locally-stored tree structures, network defenders and security researchers would encounter problems with hashes not matching – unless some hash-checking code was shimmed. Instead of modifying the data, it is more likely that these network defenders would look for the presence of these data and work to decode them to deduce the commands given to the botnet.

One ripe target for this analysis is the *application binary interface* (ABI) file, generated when smart contract source code is compiled into bytecode. This ABI file serves as a blueprint, outlining the function names and how nodes can interact with the functions. This blueprint must be distributed to each node that interacts with the smart contract (runs functions or accesses variables). The address of the smart contract on the blockchain is combined with the local copy of the ABI file as follows to gain access to the functions of the smart contract:

```
var c2 = eth.contract("c2.abi").at("0xcd1d7e71f557659ef0b2dd052d51a786f262ea6c")
```

The ABI file is a JSON-formatted list of each function and its properties. This excerpt from the botnet ABI file shows the *queryC2* function requiring no input and returning a string:

```
{"constant":true,
 "inputs":[],
 "name":"queryC2",
 "outputs":[{"name":"","type":"string"}],
 "payable":false,
 "stateMutability":"view",
 "type":"function"},
```

Figure 6. Part of an ABI file from a botnet's smart contract.

The value of an ABI file to a network defender is primarily knowing the names of each function that exist on the smart contract, and secondarily knowing the number,

types, and variable names of input and output on each function. Attempts to modify the function names in the ABI file resulted in the function becoming inaccessible - by either the old or new name.

Endpoints, when analyzed, may present information to network defenders about the location and quantity of nodes of the botnet as well as the functions of the smart contract. However, due to the hashing of the blockchain data, the instructions for the bots are protected from defensive activities such as tampering by network defenders. As with any malware, the endpoints are susceptible to traditional signature-based detection techniques (running processes, file hashes, mutex names, service names, etc.).

2.4. Stealing Private Keys

Smart contracts, like encrypted files, are only secure if the private keys that own them are kept well-guarded. If the private keys fall into the hands of researchers or law enforcement investigators, the smart contract which controls the entire botnet could be neutered or repurposed to dismantle the botnet. Although the bot herder might want to use a variety of computers when managing the botnet to hide his IP address and location, he increases the risk of inadvertently handing over the private key each time he uses it from a different node of the private blockchain.

In the geth client, private keys are stored in the *keystore* folder of the client's root, or "data directory", of the private blockchain. The keystore is a simple plaintext JSON file which is easily copied. The components of the JSON keystore file include the ciphertext, algorithm used, and key derivation function (Maffre, 2017). Here is a sample keystore file:

```
ubuntu@Ether2:~$ ls -l c2/keystore/
total 8
-rw------- 1 ubuntu ubuntu 491 Jul 27 01:08 UTC--2018-07-27T01-08-36.712311430Z--29ce889a066ba84fd5ed26275ac58497faad9f08
-rw-rw-r-- 1 ubuntu ubuntu 492 Aug 14 02:21 UTC--2018-07-31T04-16-20.094934936Z--b2ebd03aad1d9d6118640b928281bfc9670f6cd3
ubuntu@Ether2:~$
ubuntu@Ether2:~$
ubuntu@Ether2:~$
ubuntu@Ether2:~$ cat c2/keystore/UTC--2018-07-27T01-08-36.712311430Z--29ce889a066ba84fd5ed26275ac58497faad9f08
{"address":"29ce889a066ba84fd5ed26275ac58[illegible]","crypto":{"cipher":"aes-128-ctr","ciphertext":"8f10fd64ad2111 2afbfa3
91cc349[illegible]02c633e0e0906ef7[illegible]","cipherparams":{"iv":"698e0d12b4b5fbae1779964[illegible]e"},"kdf":"scrypt","kdfp
arams":{"dklen":32,"n":262144,"p":1,"r":8,"salt":"dcb0f0cb4bdf73e13ece[illegible]75dafc40e62fe10f12be59f137eece0f19"},"mac"
:"9763f5180d88365d5f58e37e0a6bae[illegible]781429ca52f5de867d"},"id":"d1caf03e-d1ba-4b47-9bd1-acfe888f60ef","version"
:3}ubuntu@Ether2:~$
```

Figure 7. An Ethereum client's *keystore* folder on Ubuntu, showing the private keys used to interact with the blockchain and smart contract.

For a botnet researcher to make use of the keystore file, the accompanying password must be obtained as well because the key is protected by symmetric encryption. This procedure was tested in a lab environment and once the <500-byte file was copied to the keystore folder of another node, the second operator had full access to the account and smart contract. This is a very valuable method for network defenders to use to dismantle a botnet. In summary, a private key left behind by the bot herder could be used to dismantle the botnet – but only if the password is obtained as well.

3. Countermeasures

For the vulnerabilities outlined above, there are countermeasures that the bot herder might take to mitigate the effectiveness of botnet researchers and network defenders. The bot herder will include some of these properties in the design of the botnet, while others are related to his or her activities in managing the botnet.

The main countermeasure to enumerating the size of the botnet is to design the peer model to be very distributed (e.g. each node is only connected to five others). Otherwise, there is not much the bot herder can do to prevent enumeration – such as preventing defenders from analyzing network traffic patterns between peer nodes or C&C servers.

To prevent fragmentation, the bot herder would build alternate communication channels into the botnet. These would consist of backup private blockchains, secondary smart contracts, or a classic C&C method such as a domain name to query. Each of these steps is simple to implement as a backup method in the design.

To mitigate the risk of losing the private keys which manage the smart contract, the bot herder should not leave copies lying around – managing the smart contract from just one or two trusted and secured nodes. The bot herder should not use these nodes for any other personal or botnet-related activity. When migrating periodically to other nodes

hosted at different ISPs, he should securely delete the copies of the private keys first. The private keys should utilize complex passwords. Additionally, the entire contract will be replaced periodically just in case a private key fell into the hands of law enforcement. The new contract will be owned by a new account – with a new private key. The bot herder is likely to use Tor or other anonymous proxy services when using private keys on blockchain nodes to interact with the smart contract in order to protect his identity and location.

A general countermeasure that mitigates several vulnerabilities is to fluctuate the ports used for botnet communication – or masquerade as HTTP traffic on TCP port 80. This step increases the difficulty of performing network-based enumeration to detect member endpoints and of identifying suspicious activity or traffic to C&C servers. Traditional endpoint detection techniques, however, will be effective against this botnet design.

Author Name, email@address

4. Conclusion

The encryption and authentication used by Ethereum create some serious challenges to network defenders who try to dismantle smart contract-based botnet C&C communication channels. These challenges amplify those previously researched, related to the distributed resiliency of blockchains and access control of smart contracts (Sweeny, 2017). The encryption used by Ethereum makes it implausible that the smart contract or the blockchain itself could be modified successfully by researchers attempting to dismantle the botnet. Similarly, the authenticated network communication between botnet peers is not likely to be successfully modified.

The weaknesses of this model which are exploitable to network defenders include the enumeration of the clients of the botnet, swiping the private keys used for managing the smart contract, detecting endpoint indicators, and network denial of service to the blockchain nodes. While these weaknesses do present some risk to the bot herder, the benefits of smart contracts on private blockchains outweigh them – implying that bot herders will likely look to use private blockchains as a reliable method to conduct command and control.

Author Name, email@address

References

Buterin, V. (2015, November 15). *Merkling in Ethereum*. Retrieved from Ethereum Blog: https://blog.ethereum.org/2015/11/15/merkling-in-ethereum/

ConsenSys. (2018). *Ethereum Dissectors*. Retrieved from Github: https://github.com/ConsenSys/ethereum-dissectors

Gerring, T. (2016, January 22). *What is Swarm and what is it used for?* Retrieved from StackExchange: https://ethereum.stackexchange.com/questions/375/what-is-swarm-and-what-is-it-used-for

Github Contributors. (2016, April 4). *Pylevel User Guide*. Retrieved from Read the Docs: https://plyvel.readthedocs.io/en/latest/user.html

Github Contributors. (2018, May 8). *Choosing A Client*. Retrieved from Ethereum Homestead: http://ethdocs.org/en/latest/ethereum-clients/choosing-a-client.html

Github Contributors. (2018, May 25). *Ethereum Wire Protocol*. Retrieved from Github: https://github.com/ethereum/wiki/wiki/Ethereum-Wire-Protocol

Github Contributors. (2018, August 14). *Patricia Tree*. Retrieved from Github: https://github.com/ethereum/wiki/wiki/Patricia-Tree

Github Contributors. (2018, March 27). *RLPx*. Retrieved from Github: https://github.com/ethereum/devp2p/blob/master/rlpx.md

Github Contributors. (2018, March 4). *RLPx Encryption*. Retrieved from Github: https://github.com/ethereumproject/go-ethereum/wiki/RLPx-Encryption

Jankov, T. (2018, June 1). *Ethereum Messaging: Explaining Whisper and Status.im*. Retrieved from sitepoint: https://www.sitepoint.com/ethereum-messaging-whisper-status/

Kripalani, R. (2018, June 21). Retrieved from Twitter: https://twitter.com/raulvk/status/1009887349721968641

Author Name, email@address

Maffre, J. (2017, December 10). *What is an Ethereum keystore file?* Retrieved from Medium: https://medium.com/@julien.maffre/what-is-an-ethereum-keystore-file-86c8c5917b97

Malaika, M., Al Ibrahim, O., & Al Ibrahim, N. (2017, November 15). *Botract: Abusing Smart Contracts and Blockchains for*. Retrieved from omProtect: https://www.omprotect.com/wp-content/uploads/2017/12/BotDraftPaper-v1.pdf

Murthy, M. (2017, December 26). *Life Cycle of an Ethereum Transaction*. Retrieved from BlockChannel: https://medium.com/blockchannel/life-cycle-of-an-ethereum-transaction-e5c66bae0f6e

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

Saini, V. (2018, August 3). *Getting Deep Into Ethereum: How Data Is Stored In Ethereum?* Retrieved from Hackernoon: https://hackernoon.com/getting-deep-into-ethereum-how-data-is-stored-in-ethereum-e3f669d96033

Sweeny, J. (2017, August 31). *Botnet Resiliency via Private Blockchains*. Retrieved from SANS Reading Room: https://www.sans.org/reading-room/whitepapers/covert/botnet-resiliency-private-blockchains-38050

Trend Micro. (2013, November 19). .*Bit Domain Used To Deliver Malware and other Threats*. Retrieved from Trend Micro: https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/bit-domain-deliver-malware-and-other-threats

Author Name, email@address

5. Appendices

5.1. Solidity Code for Botnet Smart Contract

Solidity is the most popular object-oriented programming language used to write smart contracts in Ethereum. This smart contract code was based on that provided in "Botnet Resiliency via Private Blockchains" (https://www.sans.org/reading-room/whitepapers/covert/botnet-resiliency-private-blockchains-38050) but was updated to be compatible with current versions of Solidity. The most significant change is that constructors now require the "constructor" keyword instead of using a function name that matches the contract name.

```
contract c2 {
        address owner;
        string currentVersion;
        string command; //i.e. ddos {target}, cmd notepad.exe
        string bot_url;
        string c2list;

        //Constructor is automatically executed upon creation:
        constructor() public {
                owner = msg.sender;
        }

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

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

Author Name, email@address

```
        function updateV(string v) public {
                if(msg.sender != owner) return;
                currentVersion = v;
        }

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

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

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

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

        // Current list of C2 servers:
        function queryC2() public constant returns (string){
                return c2list;
        }
}
```

5.2. Scripts to Ease Contract Deployment

These scripts were used to increase the efficiency of repeatedly testing and deploying new smart contracts to private blockchains. This first script is used to compile

the code, wrap the compiled code with instructions that Ethereum requires, and then launches geth, the "Go Ethereum" client.

```
#!/bin/bash
# Remove old files (prevents compilation errors):
rm -f c2.abi
rm -f c2.bin
# Compile the Solidity code:
solc -o . --bin --abi c2.sol
# Add required content to ABI file:
sed 's/^/var c2Contract = eth.contract(/' c2.abi | sed 's/$/\)/' >
c3.abi
# Add required content to BIN file:
sed 's/^/personal.unlockAccount(eth.accounts[0])\nvar c2 =
c2Contract.new( \n{ from: eth.accounts[0], \n data: "0x/' c2.bin >
temp.out
echo '", gas: 800000 } )' | cat temp.out - > temp2.out && mv temp2.out
c3.bin
# Launch geth in preparation to upload ABI & BIN files to blockchain:
geth --datadir c2 --nodiscover --networkid 123123 console 2>console.log
```

This (very short) second script is used to capture network traffic while the smart contract is being created and distributed. It simply saves a copy of all traffic except for the SSH session (TCP port 22) which the remote tester uses to manage the smart contract nodes.

```
#!/bin/bash
# Create a date string for use in storing PCAP file:
DATE=$(date +"%F_%H%M%S")
# Capture packets and write to PCAP file:
sudo tcpdump -ni eth0 not port 22 -w $DATE.pcap
```