SocialBotnet: A Twitter-Based C&C Channel

Jonah Varon, Jake Steeves, Liam Mulshine

***Abstract*—This paper discusses the design and implementation of SocialBotnet, a covert communication channel and steganography protocol for bot networks, which provides limited detectibility and robustness to obstruction. SocialBotnet utilizes the popular social media platform, Twitter, to covertly communicate with malware-infected, network-connected "bot" computer systems. It uses a unique image-based encoding scheme to camouflage messages as seemingly innocuous images within comments on Tweets from moderately popular Twitter accounts. Bots scan for covert messages on a range of Twitter accounts, decode the message, and execute any instructions that it contains on the infected machine.**

Keywords*—*Botnets, Covert Communication Channel, RSA Signatures

I. Introduction

With the massive growth of the Internet, the threat and potential impact of distributed botnet attacks has grown considerably more serious. Historically, botnets have consisted of a collection of malware infected machines with some centralized controller, called the bot herder, who remotely controlled the behavior of each individual bot. The herder communicates with the compromised machines within its bot network (via IRC, telnet, etc.) [1] in order to control their behaviour and execute either large scale distributed attacks (such as recent DDoS attacks), or attacks on individual local machines that expose sensitive personal information, etc. Recently, defenses against botnet formation and distributed attacks have grown more robust, largely due to improvements in the ability to detect and permanently sever suspicious communication channels between bots and their master clients. For instance, botnets that use an IRC-based command-and-control (C&C) server can be disrupted by shutting down the central IRC server [2].

To account for this higher botnet detection accuracy and central point of failure, botnet controllers have transitioned to using more covert communication methods (including encrypted peer-to-peer communication (P2P), and dynamic DNS domain based communication, etc) [2]. Recently, Botnets have begun to utilize social media platforms as communication mediums between the central C&C and distributed system of bots. For example, a C&C channel that encoded commands within Tweet text on Twitter was proposed in 2013 [3]. However, the more obvious approaches of embedding commands within text on social networking websites are easily detectable by either humans or algorithms, because they do not resemble a typical user's social network posts. These approaches also are limited by the maximum allowed text length imposed by the social networking platform. For example, Twitter limits the text of Tweets to 280 characters (as of 2017; previously the limit was 140 characters). This forces text-based C&C approaches to limit their commands to a given set of possible commands, possibly expressed as codes, which the bots must maintain knowledge of [3]. Therefore, the set of possible actions that the "bot herder" can instruct the bots to execute is limited.

Our objective is to introduce a new covert communication channel that conceals botnet messages within images on Twitter, which improves upon prior text-based approaches in both detectability and expressiveness. Our solution consists of two main components for the C&C and bots respectively. The C&C component must provide a means to embed python bytecode within an image in a manner that limits the potential for visual detection. The method that it uses to embed the message must have authenticity guarantees, to prevent outsiders from uploading their own images that force bots within the botnet to execute commands that disrupt the botnet framework. The bot component must have the means to collect images from comments on moderately popular twitter accounts, verify whether an image holds a hidden message, decode the hidden message, and verify both the authenticity of the message source and the integrity of its contents.

The process of hiding secret messages within files, images, text, audio streams, etc., is known as Steganography. On its own, steganography serves as a primitive crytography scheme, working off of the principle of "security by obscurity". While ineffective at providing absolute secrecy, this scheme helps to conceal the presence of a communication channel at all, similar to how a deniable file system conceals the presence of sensitive information on a machine. Steganography and has been employed in the past to create covert communication channels on social media sites [3, 4]. This paper focuses on the application of image based steganography to create robust botnet communication channels, that improves upon previous social media based botnet communication channels in its bandwidth, information density and limited detectability.

Given the paucity of documented research in this specific field, we needed to develop a set of metrics to use as both guidelines for our design decisions during the development process, and as points for analysis of the overall effectiveness of our implementation.

The key metrics we used:

- The amount and overall density (bits / pixel) of information that we can conceal within an image
- The visual detectability of our encoded messages from the C&C
- The expected speed with which a message might reach a bot given user usage rates and habits on social media sites
- The overall detectability of our bot on infected machines

The remainder of this paper is organized as follows. First we provide an overview of the system that we developed, consisting of two distinct programs for the C&C and bot components respectively. Then describe the design of each component in technical detail, explaining the unique encode/decode protocol that we used to achieve variable information density in our images. We present the results of our implementation, and analyze these results using the design metrics that we defined above. Lastly, we discuss some related work and areas for future research.

II. Implementation

In order to demonstrate the capabilities of this system, two programs and a protocol are required:

- A program the bot herder runs (1), which embeds botnet instructions in image files, according to the protocol. These images will be posted on various social media accounts by the attacker.
- A program the bots run (2), which periodically checks the designated social media sites and downloads, extracts and executes the hidden instructions. These instructions can change the subsequent location at which the botnet will look for instructions (for instance, changing the twitter username for replies each time).
- A protocol for the bot to parse online images for code instructions; this protocol will allow for the dynamic placement of code at any location in an image, and it will also provide a method for verifying the authenticity of the sender of the instructions.

To limit detectability, the protocol allows for each command to designate a new set of accounts to check their replies for the next encoded image; these locations could be the replies to a certain account's posts. In this way, the bots' traffic to social media sites will appear to be more similar to a typical user.

Python code is embedded into images according to our protocol. Following the same standards, the bot will decode the image and save the instuctions to a Python file which is run as a subprocess, and then the file is removed. We will describe in depth the exact mechanisms through which these processes occur.

The controller of the bot network posts these images infused with encoded instructions in the replies to designated Twitter accounts following the site's usual method. Because Twitter does not impose strict measures to verify accounts, the bot controller can trivially use a different, newly-created Twitter account to post these replies each time, to avoid detection. The bot program periodically monitors the replies to tweets from accounts specified in a list, searching for an image that contains a valid identifier. Once such an image is found, the bot decodes the message using our protocol, which we describe in section II.B. The bot verifies the message's authenticity and integrity via the C&C's RSA signature, and then interprets the received message, executing any instructions included. We made a simple listener program to test a proof of concept for this protocol as an efficient and inconspicuous method of covert communication over a public forum.

The protocol that we developed to encode data into images optimized over two parameters: information density, and visual detectability. The process consists of two distinct components: (1) converting a malicious script to binary bytecode destined for execution on bot computers, and (2) encoding the bytecode bit by bit in each image pixel color channel.

Our decision to use 4-channel .png files for C&C message encoding was not arbitrary. This file format has two key advantages over related file formats, particularly 3-channel .jpeg and .png files. A 3-channel image typically holds three bytes of information per pixel, indicating the intensity of red (R), green (G) and blue (B) light within each pixel. A fourth transparency (α) channel can be added to .png files, which effectively adds regional transparency to an image. To our benefit, this additional layer increases the storage potential of our image based encoding scheme by at least $\frac{1}{3}$. More importantly, however, it prevents Twitter from converting the image to a .jpg and passing it through its usual compression algorithms, which would render any hidden message unintelligible to the bot network.

A. *Converting Instructions to Bytecode*

In order to reliably send Python code as binary data, we prepend metadata to the compiled Python bytecode that indicates the length of the bytecode and proves that the code comes from a trusted source. The resulting payload for our protocol has the following format, where each section is represented in binary:

[initiator][RSA signature][code length][Python bytecode]

Every valid bytecode instruction will start with the binary representation of an initiator sequence as specified in the configuration file of the bot network. When creating the bytecode, the bot herder must have the same initiator sequence specified in their local configuration file. Characters are represented as 8 bit binary strings (with the most significant bit first) according to their ASCII code, and these strings are in the same order as the original text. In this manner, the initiator sequence and the Python script are converted into binary data. The length of the code in bytes is converted to a binary number with padding to a maximum of 32 bits. These fields allow bots in the bot network to dynamically locate the bytecode sequence encoded in an image, finding its start location with the specified initiator and its end based on the instruction length.

Since our images are posted on public forums, an RSA key pair prevents other parties who know the secret identifier from sending their own instructions over Twitter. When the bot herder creates the bytecode for a script, they sign the binary instructions with a private RSA key, so that the bot network can use the associated public key to verify the source before executing the decoded instructions. These keys are generated through OpenSSL, and the OpenSSL digest commands are used to sign and verify data. After the code is converted to binary and signed with the private key, the output is transformed to binary in the same way and placed between the initiator sequence and the length. The signature is a constant

256 bytes, and the length is a 32-bit (4-byte) integer, so bots can easily find the start of instructions after they have dynamically found the end of the identifier.

B. Embedding Code in Images

After breaking the malicious Python script into bytecode, and getting the corresponding binary representation, the command and control server must now encode the binary attack script with the authenticating metadata into a seemingly innocuous image file for upload to an unsuspecting Twitter account. The encoding scheme that we devised allows the C&C to make a tradeoff between visual detectability and message information density. We will now discuss our encoding scheme in detail.

Images provide a good hidden communication channel, because of the density of information that they can pack and their ostensible innocuousness. For example, a 200x200 color image (.png) with three one-byte color channels (R, G, B) and a fourth transparency/alpha channel can convey 160KB of information. One naive encoding strategy that achieves maximum information density replaces each color/transparency channel in the original image with bytes from the python bytecode. Unfortunately, this completely clobbers any visual information that the original image was attempting convey, leading to high visual detectability.

A second, rather naive approach minimizes detectability but achieves poor information density. In this approach, each image color channel byte encodes a single bit from the bytecode binary. Each byte is altered by at most 1 so that the parity of the color channel byte matches the value of the corresponding bytecode binary bit (an even byte corresponds to a 0 and an odd byte corresponds to a 1).

Our implementation permits both of the aforementioned approaches; however, it introduces two new encoding densities, that provide a nice trade-off between detectability and information density. This approach allows the C&C server to specify the number of bits, N, of information to encode in each image pixel color channel from the set, $S : \{1, 2, 4, 8\}$. The two extremes, $N = 8$ and $N = 1$, correspond to the first and second naive approaches described above.

The exact algorithm used to encode N bits from the python bytecode per color/transparency channel utilizes the mod operator. Each channel is tweaked such that it is "mod N" equal to the corresponding N bits from the python bytecode. For example, suppose we were using an $N = 4$ bit encoding scheme and needed to encode the following stream of bits from the python bytecode binary:

0101 1111 1010 1101

We have split the bit stream into groups of 4 for convenience. The first group of 4 bits will be encoded in the first image pixel's red "R" channel. Suppose that this pixel's "R" color channel takes on the value, $c \in [0, 255]$. Then, c', the new color channel value that will hold the message, $v = 0101_2 = 5_{10}$ is defined by

$$c' = (p - p \cdot mod(2^N) + v) \tag{1}$$

The next three groups of $N = 4$ bits from the bit stream will be placed in the pixel's "G", "B" and "alpha" channels respectively, using the same encoding scheme as above. Subsequent N bit chunks will be encoded in subsequent pixel color channels. Please note that care was taken near the upper bound of 255 to ensure that no pixel color/alpha channel was mapped to a value outside the acceptable set of $[0, 255]$. Therefore, if $c' > 255$, c' was instead updated with this algorithm c' = (p - p mod (2N) - (2N- v))

$$c' = (p - p \cdot mod(2^N) - (2^N - v)) \tag{2}$$

C. Decoding Images

The process of decoding messages from images that have passed through our encoding protocol proceeds as follows:

1) Determine that the image contains a message from our C&C
2) Decode the message, and identify three key fields: RSA signature, message payload length, and message payload
3) Verify the authenticity of the message and the integrity of the data that it contains using the message payload, the C&C's public key, and the RSA signature

Now we describe in detail each of the steps enumerated above. After collecting a set of images from the targeted Twitter accounts, the bot identifies which images contain encoded information from our C&C. This is done by decoding the first few pixels within the image and comparing the resulting values with the known initiator sequence, defined in the botnet configuration ("pr0blematic" in our implementation). Note, however, that the bot has no knowledge of the encoding density used to encode this message and must decode the first few pixels within the image at each permitted encoding density, $N \in \{1, 2, 4, 8\}$ until either the "pr0blematic" keyword is identified, or all encoding densities have been attempted. In the latter case, the image is deleted, since it contains no relevant information.

The decoding process involves applying the mod operator to each pixel color channel. For example, the i^{th} N-bit group, m_i, encoded in color channel c_i is

$$m_i = c_i \cdot mod(2^N) \tag{3}$$

This decoding scheme is applied to each color channel of each successive image pixel, and the result is appended to the results of previous decoding iterations. After all pixels used to represent the message have been decoded, the original message is recovered.

The bot must be able to verify both the authenticity of the message, and the integrity of the data that it contains. As explained in section II.A, the C&C authenticates itself and permits integrity checking by appending an RSA-2048 signature after the identifier. After identifying the the keyword, "pr0blematic", the bot decodes the 256-byte RSA signature from the image (using the previously identified encoding density) and stores the result to be used for verification after the message has been decoded. As explained in section II.A, the next 4 encoded bytes specify the length of the message payload. The bot must decode these 4 bytes at the appropriate decode density and convert the resulting binary value to the corresponding message length.

Once the signature has been identified and the message length found, the bot can easily decode the message payload using the decoding scheme defined in equation (3). After decoding the entire message, the message authenticity and integrity is verified by comparing the decoded RSA signature with the hash of the message generated using the public key. Assuming the verification is successful, the bot then executes the decoded Python bytecode.

D. Bot Malware

Since most users' Tweets are publicly available, scraping Tweet comments and downloading images does not require any authentication. Additionally, Twitter does not implement very strict rate limiting on web requests for public tweets. Therefore, to monitor a given list of Twitter accounts and download new tweet comments, the bot program simply issues web requests using Python's urllib2 library, and extracts the relevant URLs from the HTML using regular expressions.

The bot program reads the list of Twitter accounts to monitor for instruction-containing images from a file at the location defined in bot_config.py (default: data/twitter_accounts.txt). Therefore, to change the list of Twitter accounts in order to avoid detection, the botnet controller can simply include Python bytecode that modifies the contents of that file, as described in the Script Execution section below. The bot program keeps track of each image it has downloaded by writing the image URL to a file at the location defined in bot_config.py (default: data/seen_images.txt), which prevents instructions from being executed more than once.

Since the bot program's network traffic consists solely of HTTP GET requests to a changing set of Twitter accounts, it would be difficult for a network administrator to determine that the traffic is malicious. While not the focus of this paper, there are many additional ways this traffic could be altered to further limit its detectability, some of which are described below in Future Work.

III. Results and Analysis

After implementing our covert communication system, we evaluated the encoding protocol based on the metrics defined in section I. In particular, we assessed the amount and overall density of data encoded, the visual detectability, and the potential of our script execution to perform useful complex functions.

A. Image Encoding

To evaluate the effectiveness of our covert communication channel, we tested the protocol on our local machines. First we generated large sample bytecode files, and encoded the binary representation of these files into the pleasant image of a basking dog with varying bit density. Then we generated a few statistics such as the percentage of pixels used, the amount of data encoded, and the overall mean-squared-error of the resulting image, allowing us to quantitatively evaluate the merits of the encoding scheme at various densities (figure 1). Through visual analysis we qualitatively gauged the detectability of these encoded images.

Encoding Density	Encoded File Size	Pixels Used (%)	Mean-Squared-Error (MSE)
1-bit	170 KB	97	0.55
2-bits	340 KB	97	1.46
4-bits	680 KB	97	6.53
8-bit	1.4 MB	97	96.50

Figure 1.

In the table above, the mean squared error provides a quantitative measure of the difference between between the original image and the image with encoded data. It is calculated by averaging the squared difference between corresponding pixel color channel values over the entire image. We found that a mean squared error of less than 5 was imperceptible to the viewer, even when comparing directly with the original image. The MSE resulting from the 4-bit encoding scheme was high enough to demonstrate visual signs of distortion when compared with the original image. However, when viewed independently of the original image, the unsuspecting viewer would have no reason to believe that the image was holding any malicious information.

These examples demonstrate that it is incredibly difficult to perceive the difference between the original picture (figure 2) and the encoded version at 1-bit density (figure 3), despite the quatity of data encoded within them. Unsurprisingly, as the data density increases, the image becomes more distorted. These additional images can be seen in the project repository.

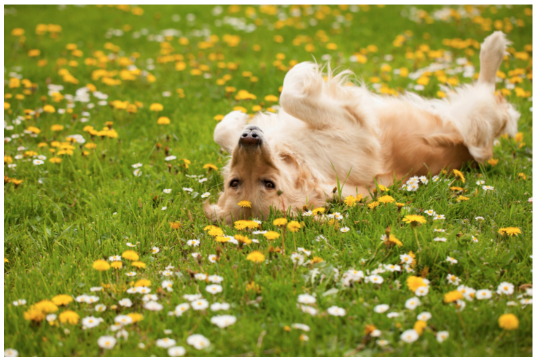
Figure 2.

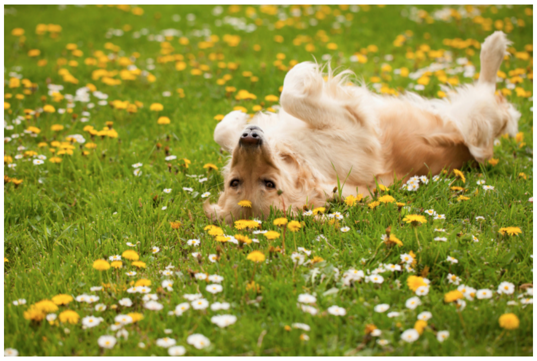

Figure 3.

The results demonstrate that the effects of our encoding scheme are nearly imperceptible to the human eye using an encoding scheme as high as 4-Bits. The naive encoding scheme, with N = 8, clobbers the original image as expected and, thus, would appear quite suspicious. We would advise using the lowest encoding density possible given the image and message size constraints that you have.

B. *Script Execution*

Since our bot program simply decodes the instructions from an image and executes that code as a Python file, there is vast potential for the modular use of our protocol to accomplish dynamically decided goals.

In order to demonstrate these capabilities, we have included examples of Python code in the scripts directory of our project file. These can be transmitted through our protocol in order to accomplish basic functions. Most importantly, we can dynamically alter the list of Twitter accounts that the bots monitor so that Twitter traffic does not generate any suspicious patterns. The information is stored simply in data/twitter_accounts.txt as a newline separated list of Twitter handles. With only 10s of lines, Python scripts allow bots to add to, remove from, or completely change this file. As long as the bot herder carefully tracks the currently monitored accounts, redefining the list is a straightforward task.

In the same manner, settings in bot_config.py or other files could be rewritten using the standard file io, and the protocol would still function properly as long as the bot master tracked the changes. A bot herder could launch updates to the public RSA keyfile or even change the initiator string. Information can be easily changed if it is every compromised by a third party who may attempt to hijack the bot net. However, as with the account list, if the controller does not carefully record changes to the system, they would be unable to send verifiable bytecode and would lock themselves out of the network.

Another useful function is demonstrated in the exfiltration script, which shows the ease with which sensitive system data can be transmitted to a dynamically defined server. The server uses simple flask code to log HTTP encoded messages that are sent via GET requests in the msg variable to the route /log?msg=[encoded-payload]. The index of the server displays all logged messages, including the IP address of each request and the time at which it was received. This basic setup serves to demonstrate the simplicity of exfiltration. Using only standard libraries the script determines the username, the home directory, and the host of the infected bot, and then it uses urllib to encode the message and send it to the server. Since the URL of the server is defined in this script, this process obviates the need for hardcoded addresses in the bot malware, and it allows the bot herder to easily switch exfiltration methods.

See the README.md within our project directory for specific usage instructions.

IV. Related Work

Pantic and Husain developed a covert communication channel specifically for Twitter, but they focused on text usage instead of image posts [4]. They encode secret messages in the metadata of tweets, using the character lengths of a set of tweets. At the time of their writing, they could encode 7 bits of data in a 140 character tweet, but despite the increase to 280 characters, the density of our protocol is much higher. A single tweet could contain multiple images, each with an exponentially higher amount of data than a single tweet in the Stego System. In order to produce a vast amount of tweets, they created a complicated program to generate innocuous messages, while our system simply requires a single inconspicuous image in order to embed an entire instruction sequence.

SecreTwit [3] is a Steganography tool that encodes hidden messages within Twitter posts (text, urls and images). It uses the LSB method to encode hidden messages within images, which is comparable to our encoding scheme with $N = 1$. Ning and Jianxia have performed in-depth analysis of steganography techniques on the JPEG file format, and the effectiveness of communicating these covert messages on social media sites [6].

V. Future Work

The process of verifying the RSA signature of bytecode data currently requires the openssl commandline functions that are commonly found on Linux systems. For the bot herder that creates the bytecode, this is not a problem, because the program can be run on any computer that the attacker controls. However, verification of the signature occurs on the bot network, which means that infected machines must currently be Linux operating systems that have openssl. Also our malware requires an infected Linux machine in order to run the Python code, but in the future this protocol could be generalized for other systems. Given enough time, these features could be implemented in lower level code, such as C, but the purpose of this project is to demonstrate the ability of our protocol to covertly communicate general code through images in a public forum.

It would be worthwhile to do an in-depth analysis of how well our bot program camouflages itself within the affected users' machines. This would require measurement of two metrics: (1) data on typical user usage rates on social media sites and (2) the frequency and intensity of high CPU usage on a user's machine. Analysis of both the frequency and content of Twitter GET requests could be done and compared with the frequency and content of similar requests made by our bot program, and improvements such as randomizing the timing of the bot malware's requests or the User Agent request headers could trivially be made. Additionally, an analysis of the CPU usage of the bot program could be done and compared with typical user CPU usage. Extreme deviations from the norm of either field (frequency/content of social media usage and CPU demands) would make our bot susceptible to detection by anti-virus software.

We leave open to future research the application of this protocol with more generalized and more advanced malware techniques for infection and masking visibility.

VI. Conclusion

This paper describes an implementation of a new covert communication channel for botnets. SocialBotnet improves upon previous methods of covert C&C communication with bots in a botnet by camouflaging both the messages from the C&C within Twitter comments on moderately popular Twitter accounts, as well as bot behavior on infected machines that mimics a user's typical network traffic. The protocol defines an effective image-based encoding scheme that can convey as high as 4 bits of information per image pixel color channel, or 16 bits per pixel, while maintaining limited visual detectability. This encoding density far surpasses the encoding density achieved by previous attempts at using social media platforms as channels for covert botnet command and control. Since our communication channel is information-dense and relatively high-bandwidth, botnet frameworks based on SocialBotnet can avoid using more conventional high-bandwidth communication channels (IRC, dynamic DNS, etc.) that are either easily detected, easily shut down or both. By supporting high C&C information density, the botnet itself can be more flexible and mutable, allowing the bot herder to alter its behavior, add capabilities, and even remotely rewrite the botnet software on the fly. In total, SocialBotnet is capable of conveying complex instructions to a wide network of bots using a fairly simple protocol that is difficult to detect.

Acknowledgment

Thank you, Professor Mickens, for a great course with some laughs along the way...

References

[1] E. Cooke, F. Jahanian, and D. McPherson, "The zombie roundup: Understanding, detecting and disrupting botnets." *SRUTI Workshop*, 2005.

[2] B. Aslam, P. Wang, and C.C. Zou, "Peer-to-peer botnets," in: *Handbook of Information and Communication Security*, Eds. M. Stamp and P. Stavroulakis. Springer, 2010.

[3] Google Code Archive. Google, Google, code.google.com/archive/p/secretwit/.

[4] K. Ross, A. Singh, M. Stamp, and A. Toderici. "Social Networking for Botnet Command and Control." *I. J. Computer Network and Information Security*, 2013.

[5] M. Husain and N. Pantic. "Cover Botnet Command and Control Using Twitter." *ACSAC*, 2015.

[6] Ning, Jianxia, et al. Secret Message Sharing Using Online Social Media. 2014 IEEE Conference on Communications and Network Security, 2014, doi:10.1109/cns.2014.6997500.