Tweetnet: Finding the Bots in the Flock

Josh Blum (joshblum@mit.edu)
Bennet Cyphers (bcyphers@mit.edu)
Ofir Nachum (ofir@mit.edu)
Louis Sobel (sobel@mit.edu)

6.857 Final Project

May 14, 2014

Abstract

Due to Twitter's scale, anonymity, and the reliability of its messaging service, it is an attractive platform to support botnet command and control. The goal of our project was to examine how such a botnet might be implemented and to determine whether such botnets could be detected efficiently. Our approach was to design a game which captures the essential elements of the botnet communication problem. In the game, two teams implement a botnet and each has a limited amount of time to send a random message over a mock Twitter network. The goal is to avoid detection by the other team while discovering the details of the other team's botnet. We ran two rounds of the game under different rules. Based on the results, we conclude that an adversary who takes sufficient design precautions can communicate complex, arbitrary commands over Twitter in a way that is most likely infeasible to detect, even if the source code is compromised.

1 Introduction

Botnets represent some of the most persistent threats on the internet today. Over the past two decades, the use of networks of hijacked computers for malicious purposes has grown tremendously. A botnet typically consists of a collection of machines infected with a particular malware program which allows the machines to covertly communicate with other infected machines, perform their intended function, and act on commands issued by a central operator. Botnet operators can leverage their armies of "zombies" to distribute spam, spread malware, or execute targeted DDoS attacks. Most users of infected computers do not know that they are compromised.

A central aspect of botnet design is the "command and control" (C&C) scheme. The operator must be able to control infected machines in a reliable and scalable manner while avoiding detection and preventing intervention from competing parties. At its core, C&C requires a way to secretly send messages to thousands of infected machines.

Twitter is an easy-to-use social media platform with a quick signup process, a reliable distributed messaging service, and a robust API. In this project, we looked at the use of Twitter as a platform for botnet C&C. Our ultimate goal was to detect a botnet "in the wild". Along the way, we wanted to determine how such a scheme might be designed, the challenges the designers would face, and what methods could be used for detection. In order to do so, we designed a game where we split our group into two teams, `bennyblum` and `louisofir`, and competed to try to communicate a message from a "master" to a collection of "slaves" on a simplified mock of Twitter. The result is Tweetnet.

2 Related Work

Botnets using social networks for C&C are a relatively recent phenomenon. These botnets take advantage of the reliable infrastructure and easy-to-use APIs of social networks to communicate commands to infected computers. Several of these botnets have already been detected.

Several botnets have been discovered in the wild [1, 2, 3]. In 2009, Arbor Networks [1] discovered a Twitter bot which periodically tweets base-64 encoded commands that specify URLs for downloading malicious binaries. The Mehika botnet [2] is another botnet which was found using Twitter to send commands to its botslaves. The Macintosh Flashback malware provides yet another example of a botnet using Twitter for C&C [3]. The malware queries Twitter for tweets containing specific hashtags.

In addition to these detected botnets, several groups have designed and implemented social networking botnets [4, 5, 6]. A group of San Jose University researchers designed and implemented their *SocialNetworkingBot* to issue commands via Twitter for such things as browsing a URL, taking a screenshot, shutting down the computer, or changing the botmaster [4]. Another notable example is *Stegobot*, a botnet which covertly communicates via steganography within images shared on social networks. These examples highlight the ease with which it is possible to use social networks as botnet infrastructure.

In response the the growing threat of botnets using social networks, researchers have proposed several detection mechanisms to thwart such botnets [7, 8, 9]. Gu et al. proposed an approach using network-based anomaly detection [7]. These methods are useful when botnet activity is temporally correlated. Kartaltepe et al. presented a content-specific method which distinguishing natural text from base-64 or otherwise encoded commands [8]. The group also suggests client-side detection mechanisms that are alerted by network traffic of dubious origin and purpose.

The previous work in this field shows that while botnets using social networks is a relatively recent phenomenon, the threat of even better concealed botnets is growing rapidly. For this reason, it is imperative that the capabilities of botnet designers be better understood and new detection mechanisms be devised.

3 Game Overview

This section describes the simulation of Twitter we used in order to experiment with bot detection and development techniques.

Twitter processes thousands of tweets per second. Attempting to identify a botnet would be extraordinarily difficult. Instead, we decided to use a mock of Twitter and structure our project as a game.

3.1 Description

We split up the four person group into two teams. Each team would take a turn designing a bot and using it to send information across the mock Twitter infrastructure. The other team would use the resulting stream of tweets to try and determine what the C&C protocol was and which handle was the botmaster. The defending team would choose what information was transferred and would know the content, timestamps, and username of all tweets.

3.2 Justification

The use of a game model for security is a reasonable decision. First, it allows simulation of tactics at a small scale. The small scale let us to focus on strategy and cryptography without having to deal with Twitter's massive data stream. Second, there is precedent for such models of security, such as IND-CCA, IND-CCP, and others. In these game models, an adversary is given generous capabilities, such as a ciphertext oracle. If a scheme is secure under these conditions, it can then be considered secure under more relaxed constraints. Our choice of allowing the defending team to choose the information the botmaster needs to transfer was guided by this precedent.

3.3 Infrastructure

We chose not to play our game actually using Twitter's services, but rather to build a limited subset of them for ourselves. There were three reasons for this. First, we were worried about the ramifications of using automated Twitter accounts on Twitter, including ones that were designed for botnets. Second, some infrastructure would be needed for the game, such as keeping track of when information has been successfully transferred. Since some infrastructure would already be needed, it made sense to create a limited version of Twitter as well. Finally, it made testing and using the bots much more efficient. We didn't have to deal with millions of tweets; we only had to deal with those germane to our game. The infrastructure we built supports basic usernames and tweeting. The code is available on Github[1].

3.4 Rules

Having given the background on our game, this section now gives a description of the rules.

- There are two teams: the attacking team and the defending team. The attacking team is trying to use the mock Twitter as a C&C channel, while the defending team is trying to catch them.
- There is a set number of mock Twitter users. Some handles are *benign* tweeters, which simulate regular people, and the other handles are controlled by the attacking team.
- The defending team generates random 128 bit hexadecimal *flags*.
- The attacking team must pass generated flags to its *bots* using *only* mock Twitter as the communication channel.
- When a bot receives a flag, it submits it to the defender as proof of successful communication.
- Bots and botmasters may have arbitrary hard-coded constants for use as shared keys.
- The botmaster has access to a *random tweet oracle*, which is guaranteed to provide a unique and realistic-looking tweet.

For enforcement of the rules, we relied on the infrastructure described above and the implicit cooperative understanding between the teams, who would not want to be labelled CHEATERS in a project.

[1]`https://github.com/joshblum/tweetnet`

4 Round 1

This section describes the setup, designs, attacks, and takeaways from the first round of gameplay, for which each team built a botmaster.

4.1 Setup

During Round 1, each team designed a single `Bot` and `BotMaster` pair which had to submit a single flag. There were 10 unique handles used. 1 was the botmaster, and the other 9 were benign users, which would tweet with a probability of 0.05 each second. `bennyblum`'s bots knew the botmaster handle at the beginning of the round whereas `louisofir`'s bots established the botmaster as part of their protocol. The goal of this round was to allow the teams to familiarize themselves with the game infrastructure and begin building detection techniques. For this reason, we allowed many simplifications to the game structure for this round, with the intent of modifying the game structure in the subsequent round.

4.2 Designs

4.2.1 `bennyblum`

`bennyblum` built the `SingleCharBotMaster` which relied on Python's pseudo-random number generator (`prng`) to signify tweets which were information carrying. The `prng` output stream was used to determine the offset of a single character in each tweet that was information carrying. The botmaster and bots both shared a common seed value for the `prng`. Figure 4.1 shows an example of how the `prng` was used to offset information tweets, and Figure 4.2 shows how this offset is used to position a character of the flag within a tweet.

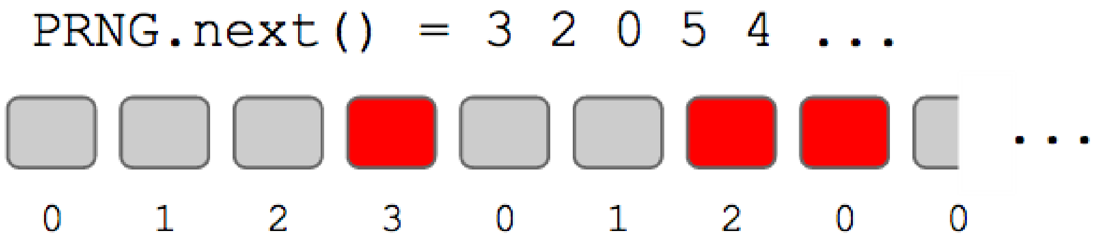

Figure 4.1: An example of how the `SingleCharBotMaster` design uses a `prng` to offset which tweets carry information.

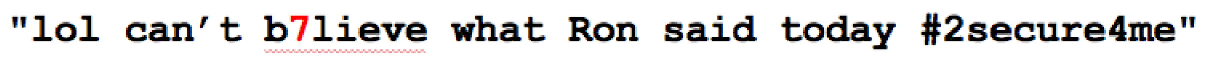
"lol can't b7lieve what Ron said today #2secure4me"

Figure 4.2: An example of how the `SingleCharBotMaster` design positions flag information in a information carrying tweet.

4.2.2 `louisofir`

`louisofir` built the `SpaceBotMaster`. This botmaster used a space at the end of tweet text to signify a its flag relevance. The master would insert a punctuation mark immediately preceding the space ('.', ',', '!', or '?'), providing two bits of the flag. Two spaces at the end of a tweet signal the start of a flag. Figure 4.3 shows an example of a flag-relevant tweet.

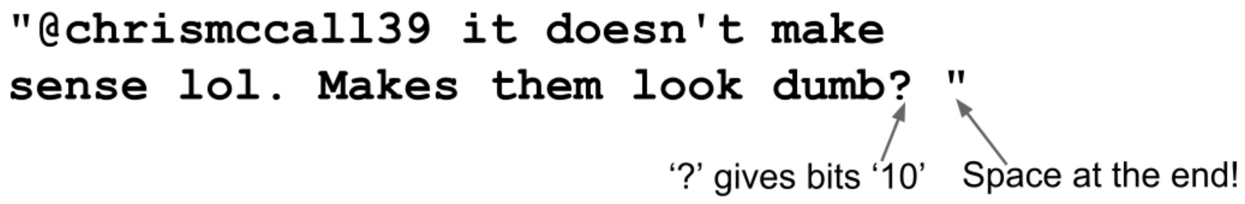
"@chrismccall39 it doesn't make sense lol. Makes them look dumb? "

Figure 4.3: An example of how `SpaceBotMaster` communicates the flag.

`louisofir` also put in capabilities for the master to identify itself. The master partitioned time to 12-second windows. When the master tweeted randomly (when it did not have a flag to communicate), it would tweet only in the first 6 seconds of a 12-second window. This way, when the botslaves analyzed user's tweets, they would notice that one had a significant amount of tweets in the first 6 seconds of every 12-second window. They would know this was the master. Unlike the master, any other benign would tweet equally in the first 6 seconds as in the last 6 seconds. In this way, the master is able to discreetly make itself known to the botslaves.

4.3 Attacks

4.3.1 `bennyblum`

The first step in analyzing the data from `louisofir` was to see if there were any inconsistencies from the data set that caused a particular handle to differentiate itself. Using the `json` Python module and simple queries across the dataset, the botmaster handle was able to be identified since the frequency of tweeting was twice that of any other handle.

bennyblum looked at several techniques in order to identify the master before the handle was found. For example, looking at the frequency of different emoticons and any signal hashtags that may have been used. Once this handle was discovered, bennyblum wanted to dig deeper into reverse engineering the protocol used.

Among the set of tweets from the suspect handle, the first thing that stuck out was that many of the tweets had a space character appended to the end of the tweet. This furthered the suspicion that the handle was the botmaster. bennyblum believed that a binary string was being encoded based on whether the tweet had a space or not.

Since bennyblum knew that the botmaster handle was not known to the bots initially, the botmaster would have had to have been established and then the 32 characters of the flag transferred. They tried various transformations of the binary string to convert it to the flag output, without success.

4.3.2 louisofir

louisofir was able to almost entirely discover bennyblum's design. They began with visual inspection of the collection of tweets, and immediately noticed aberrations such as r8ally and RTa@coolman123 – patterns that did not look anything like what would normally appear on Twitter. louisofir further noticed that all of these strange aberrations came from the same username: tweetnet01.

Then, they computed tweet frequency by username, and discovered that tweetnet01 had tweeted almost twice as many times as the rest of the usernames had. This cemented their suspicion that tweetnet01 was the botmaster. By visually inspecting all of tweetnet01's tweets with a highlighter and pen in hand, louisofir was able to find an entire flag embedded in outlying tweets. By examining the pattern of flag character offset and gap between flag tweets, they further identified bennyblum's pattern of *n*-down, *n*-over. louisofir decided this was based on a prng, and did not attempt to discover the seed.

4.4 Problems

From this round we discovered problems both with the game and bot design. The game used for this round was an inaccurate representation of Twitter. We allowed many assumptions, such as a very small user base, no social graph interactions, and access to the random tweet oracle. These simplifications gave the attacking teams a great advantage when trying to analyze the bots. For example, any analysis that was done by hand would be unrealistic in the actual Twitter environment without large machine learning or crowd sourcing efforts.

The overarching problem with both bot designs was that the botmasters' tweets prevented them from blending in with the benign tweeters. This showed how difficult steganography can be. Neither team was able to effectively mask the information they wished to hide. These types of errors were coupled with implementation flaws, such as failing to mask the flag content in the `SingleCharBotMaster` and misconfiguring the tweeting probability (0.5 was used instead of 0.05).

We did learn, however, that a high throughput is not necessary for the success of a botmaster. Since many users tweet with a high frequency, even if each tweet from the botmaster relays less than a single bit, not many tweets are needed to communicate with the bots. For example, a botmaster could post a large amount of malicious content (spam) to a public service such as pastebin [10], and simply use Twitter to communicate the URL to the bots.

5 Round 2

This section is a description of the events of the second round.

5.1 Modifications

For the second round we introduced several modifications to the game structure. First, the number of bot handles was increased to 20, 2 of which were botmasters. This increased the pool of active handles and gave the designers more flexibility. The second major change was that the code of the bots was to be open sourced for the detection team. The bots were allowed a configuration file with predefined constants that could remain remain secret, but all other code was going to be revealed. The motivation behind this was that botnets can be discovered and their source or binary de-obfuscated. The network should be robust to such an attack, and the botmaster handle should be able to remain unknown.

5.2 Designs

5.2.1 `bennyblum`

In the second round, `bennyblum` built the `HashPadBotMaster`. This bot begins by generating a secret 32-character hex string (the pad) to mask the flag, and a single hexadecimal start signal character. The start character and the pad are generated using an instance of Python's `Random` object, which was seeded with a 32-bit secret shared among all bots in the network.

When the round starts, the botmaster tweets randomly until it receives the flag from the server. Upon receiving the flag, the botmaster sends the start character followed by the 32 characters of the flag XORed with the characters of the pad. Each masked character, c_i, is sent in one tweet, chosen at random from a precomputed library of legitimate tweets such that $c_i = \texttt{hash(tweet)} \pmod{16}$. The botmaster does not send any tweets which hash to the start character until it is ready to send the flag. At all times the master tweets at the same frequency as the legitimate tweeters.

The listeners consume and discard tweets from the botmaster handle, which they know ahead of time, until one of them hashes to the precomputed start character. At this point, each bot consumes the next 32 tweets, decoding them to characters in the flag according to $f_i = \texttt{hash(tweet)} \pmod{16} \oplus p_i$, where f_i is the i^{th} character in the flag and p_i is the i^{th} character in the pad. Once all 33 tweets have been consumed, the listener can submit the flag at its leisure, adding jitter to prevent timing correlation attacks.

5.2.2 **`louisofir`**

`louisofir` built the `MD5BotMaster`. When given a flag, one of the two botmaster handles is chosen at random to communicate the flag. This botmaster first masks the flag via a shared `prng`. The 32-character masked flag, along with a prepended hexadecimal start signal, is communicated one hex-character at a time in the following way: The bot tweets a tweet with content c such that

$$d = MD5(MD5(c) \oplus S_k) \pmod{16},$$

where d is the desired hex-digit and S_k is a secret key included in the configuration file.

To further hide the flag communication, the bot does not tweet these flag-relevant tweets in immediate succession. Rather, like the original `SingleCharBotMaster`, the botmaster sends between 0 and 4 superfluous tweets between each flag-relevant tweet. The botmaster also tweets a certain number of superfluous tweets before beginning flag communication. Because of the shared `prng` seed, the botmaster and the bots will stay synchronized.

To effectively imitate natural communication frequency, `louisofir` implemented a `TweetQueue`. Tweets are pushed onto the queue. A single process handles popping and tweeting items from the queue so that a tweet frequency emulating that of the benign tweeters is maintained.

5.3 Attacks

5.3.1 **`bennyblum`**

After reviewing `louisofir`'s code, `bennyblum` decided the best attack approach was to try to see which handles, if any, distinguished themselves from the rest. Unlike the previous round, there were no implementation errors that caused obvious deviations. `bennyblum` analyzed the mean time between tweets to see if the time to find a tweet with the correct hash was noticeable.

louisofir's bots would submit the flag immediately upon decoding it fully. However, since all of the bots tweeted with the same probability, there was no way to distinguish a tweet that occurred close to the flag submission time as originating from the botmaster.

louisofir's design was very similar to the design for bennyblum. However, it did not have the implementation flaws of the HashPadBotMaster and took advantage of game features such as the random tweet oracle and constant tweet distribution among the benign handles.

5.3.2 louisofir

With bennyblum's code in hand, louisofir was able to identify the following coding errors and vulnerabilities:

1. To generate a random 128-bit integer, HashPadBotMaster generates a random float and then multiplies it by 2^{128}. However, in Python, floats only provide 32 bits of precision, so multiplying 2^{128} by a float results in the lower 16 digits as zeros. This means that a significant portion of the flag is communicated unmasked.
2. HashPadBotMaster uses a single-digit start signal. This start signal is generated by the faulty random number generator, causing it to always be 0.
3. HashPadBotMaster was intended to tweet with a certain probability every second. This was implemented incorrectly when the bot was not communicating a flag. At these times, the bot was tweeting with a certain probability every *other* second.
4. HashPadBotMaster immediately and serially communicates the flag as soon as it obtains it. Similarly, the botslaves immediately submit it as soon as they decipher it.
5. HashPadBotMaster does not use a cryptographic hash function but rather Python's built-in hash function. This hash function is not pseudorandom and actually very malleable (for example, the hash of "hello" is 840651671246116861 and the hash of "helli" is 840651671246116859).

Due to vulnerability (2), louisofir was able to narrow down the search to tweeters which only tweeted a tweet whose hash yields a '0' start signal after the first flag was created. Subsequently, louisofir looked at the 32 tweets following the start signal, and exploited vulnerability (1) to find the partially unmasked flag. This showed tweetnet16 to be the only candidate botmaster. This was confirmed via vulnerability (3): tweetnet16 had the lowest number of tweets in the period when there was no flag to be communicated.

louisofir was even able to successfully perform a brute-force a search for the randomization seed, which definitively identified tweetnet16 as the botmaster.

The other vulnerabilities were not used to find the botmaster. However, given a larger collection of tweets to analyze or more adverse conditions for the detection team, these vulnerabilities could have been exploited to better detect botnet activity.

5.4 Problems

The main problem that undermined the botnet designs in this round were not protocol issues, but rather coding errors. These are still valid issues to discuss, as actual implementation of a protocol is an important part of applied security. bennyblum was undermined by an incomplete understanding of the Python numerical model. Errors with under and overflow are a very common source of bugs. Additionally, both teams put too much faith in the default Python prng. It can easily be brute forced. The number of bits in its seed is only 32 – even when a Python long is used as the seed, Python internally uses the hash value of that long, which is only 32 bits. These issues are implementation issues, not protocol ones. Mistakes like these are a common source of security issues, and it is possible that Python is not a suitable language for designing cryptographic protocols.

6 Discussion

This section presents a discussion of the results above. We discuss what we learned, the applicability of our project to the actual Twitter service, and possible future work.

6.1 What we learned

Through our iterative game, we realized the challenges faced by botnet design and detection teams. First, we realize that tweet frequency is a key aspect of any botnet and can often be a botnet's most vulnerable component. In the first round of gameplay, both teams were unsuccessful at achieving a natural tweet frequency. This single flaw was clearly visible to botnet detection teams and made identifying the botmaster easy. In the second round, even with the lessons of the first round learned, bennyblum still made mistakes which led to detectable and unnatural tweet frequencies. This shows that making frequency of communication natural is a key goal for designers of botnets using social networks. Although this is a feasible goal, it is very easy to get wrong. From the botnet detection perspective, characterizations of natural communication frequency and tools to detect deviations from this frequency would be quite useful.

Another key aspect is natural tweet content. In Round 1, both teams used communication methods that altered tweet text. These methods were easily detected. This shows that even seemingly concealing communication tactics that change natural text (like single character alterations) are not enough to hide information. However, botnet designers can thwart this by using the hashing method used by both teams in Round 2. Despite this work around, it is still very important for detection teams to have a tool for distinguishing natural from unnatural text, as many botnet designers may overlook this vulnerability.

6.2 How this extends to real Twitter

Some of what we've learned about detecting and creating C&C channels over our mock Twitter can be applied to the actual Twitter service. One thing is the ease of hiding information in a tweet by using the hash of a randomly generated and realistic looking tweet. This technique could definitely be used to send significant amounts of information covertly. Another thing we learned that does extend to real Twitter is that for a botnet to be successful, it would not need to transfer large amounts of data over the C&C channel. Information on the order of bytes is sufficient to serve as a pointer to further instructions for the botnet. Twitter may be just one component of an elaborate C&C scheme.

6.3 How this does *not* extend to real Twitter

A related problem is that of spambots. Spambots are computer-controlled Twitter handles which tweet out ads, links, and otherwise annoying messages on behalf of a company or anonymous mischief-maker. Spam detection is difficult. Several papers have been written on techniques for identifying these bots among the large datasets that Twitter produces [11, 12, 13]. As detection techniques advance, spambots evolve to compensate. Furthermore, since the goal of spambots is to send information for human consumption, they are restricted to messages in plain text or URLs. Botnets using Twitter are not hampered by such limitations. As we have demonstrated, information can be carried by sequences of arbitrary tweets using hashing, so a botmaster only needs a sufficiently sized library of legitimate-sounding tweets to be able to transmit any possible message. Therefore, it seems likely that any botnet with sufficiently motivated operators can evade automated bot detection comparatively easily.

In our tests, there were a small number of handles providing background noise in a very predictable fashion. This made it easy to look for subtle variations in timing and content in each handle's tweets and to perform extensive analysis on each one. The real Twitter stream consists of millions of handles with wildly varying tweet patterns in several languages, so it is much more difficult to identify breaks from general trends. For example, `bennyblum`'s botmaster in Round 2 was identified in part because it tweeted half as frequently as normal handles. In reality, the normal range of tweet frequency spans orders of magnitude (anywhere from several times an hour to once or twice a year), so it would be unrealistic to use this metric alone to identify bots. In addition, it is computationally infeasible to analyze any significant fraction of Twitter's entire tweet stream at the level of detail that we were able to.

6.4 Future Designs / work

The current mock Twitter infrastructure could be extended to modify the gameplay and make the simulation more realistic. Some aspects of gameplay were never explored, such as analyzing the social graph through followers or having a distribution of benign user behavior. Modifying these parts of the infrastructure would change how the detecting teams could perform their analysis and move the detection closer to what it would be like when using real tweets.

Another feature that could be modified is the botmaster's access to a random tweet oracle. This assumption allowed the botmaster to generate realistic tweets on demand, an ability which is quite complicated in real life. Forcing botmasters to develop their own tweets (or only retweet those of other users) would make the job of the botmaster much more difficult. It would be a realistic challenge that a botnet would face if they wished to hide their C&C channel within Twitter.

7 Conclusion

Botnets are hard to build. In any implementation, there is a good chance that subtle errors in design will lead to identifiable patterns in the output. However, if enough security precautions are taken, we believe it is possible to design a cryptographically secure scheme that is very difficult to detect. The volume and variety of data in Twitter's stream are also obstacles to detection. Any detection mechanism deployed on Twitter will have to be efficient and scalable. Some of the methods we used were not. Regardless, we produced a good framework for modeling the use of Twitter for the command and control of botnets.

8 References

[1] J. Nazario, "Twitter-based botnet command channel." `http://www.arbornetworks.com/asert/2009/08/twitter-based-botnet-command-channel/`.

[2] R. Romera, "Discerning relationships: The mexican botnet connection." `http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp_discerning-relationships__mexican-botnet.pdf`.

[3] P. James, "Flashback mac malware uses twitter as command and control center." `http://www.intego.com/mac-security-blog/flashback-mac-malware-uses-twitter-as-command-and-control-center/`.

[4] A. Singh, A. H. Toderici, K. Ross, and M. Stamp, "Social networking for botnet command and control.," *International Journal of Computer Network & Information Security*, vol. 5, no. 6, 2013.

[5] Trojan7Malware, "Botnet using twitter as the c&c." `http://trojan7malware.blogspot.com/2013/06/botnet-using-twitter-as-c.html`.

[6] S. Nagaraja, A. Houmansadr, P. Piyawongwisal, V. Singh, P. Agarwal, and N. Borisov, "Stegobot: a covert social network botnet," in *Information Hiding*, pp. 299–313, Springer, 2011.

[7] G. Gu, J. Zhang, and W. Lee, "Botsniffer: Detecting botnet command and control channels in network traffic," 2008.

[8] E. Kartaltepe, J. Morales, S. Xu, and R. Sandhu, "Social network-based botnet command-and-control: Emerging threats and countermeasures," in *Applied Cryptography and Network Security* (J. Zhou and M. Yung, eds.), vol. 6123 of *Lecture Notes in Computer Science*, pp. 511–528, Springer Berlin Heidelberg, 2010.

[9] P. Burghouwt, M. Spruit, and H. Sips, "Towards detection of botnet communication through social media by monitoring user activity," in *Information Systems Security* (S. Jajodia and C. Mazumdar, eds.), vol. 7093 of *Lecture Notes in Computer Science*, pp. 131–143, Springer Berlin Heidelberg, 2011.

[10] "Pastebin." http://pastebin.com/.

[11] Z. Chu, S. Gianvecchio, H. Wang, and S. Jajodia, "Detecting automation of twitter accounts: Are you a human, bot, or cyborg?," *Dependable and Secure Computing, IEEE Transactions on*, vol. 9, no. 6, pp. 811–824, 2012.

[12] A. H. Wang, "Detecting spam bots in online social networking sites: a machine learning approach," in *Data and Applications Security and Privacy XXIV*, pp. 335–342, Springer, 2010.

[13] G. Stringhini, C. Kruegel, and G. Vigna, "Detecting spammers on social networks," in *Proceedings of the 26th Annual Computer Security Applications Conference*, pp. 1–9, ACM, 2010.