CS 4404 Mission 3

Matt Puentes, Cole Granof

December 2018

1 Introduction

At one time, botnets were one of the biggest threats to the Internet as we know it today. With their huge scale and ease of creation, they put server-destroying power into the hands of everyday internet users. While this concern is mitigated today by the widespread use of modern security features such as the firewall, botnets remain a real threat to the Internet today. A well-known example was the Mirai botnet in 2016, which infected internet of things (IOT) devices and had a peak size of 600k devices (Antonakakis et al. 2017). The reappearance of botnets in modern times and the surge of low-security computers such as those found in IOT devices shows that the computer security community can't just look at preventing botnet infection anymore. Rather, there needs to be a focus on detecting already existing botnets by analyzing how they communicate. This paper will attempt to do just that, firstly by creating a novel form of advanced persistent threat (APT) communication, and secondly by developing an extensible defensive strategy to be used against this and similar APTs.

For this paper, we assume the existence of a game "Counter-Shue: Global Election". This game is assumed to have some sort of security vulnerability, allowing an attacker to modify the program to run an attack. This game is also assumed to be structured similarly to the real-life example of "Counter-Strike: Global Offensive." This real-world parallel uses UDP data transmission with a variable packet size (Claypool, LaPoint, and Winslow 2003), so we structured our cover traffic around the data found in the paper cited above.

2 Reconnaissance

APT attackers have several concerns when it comes to the command and control (C&C) infrastructure of their attack. The fact that running a botnet is illegal means that they have to be incredibly careful, as getting caught can result in serious jail time. This need to be covert also helps ensure the standard security needs: Confidentiality, Integrity, Availability, and Authenticity.

Confidentiality is obviously important due to the inherent risk of running a botnet, but it also goes beyond just keeping the identity of the botnet user hidden. Confidentiality ensures that an outside observer shouldn't be able to tell what is being said, but it also keeps them from knowing that communication is happening at all. This is related to the need for an botnet attacker to utilize covert communication. A common way botnets try to achieve this goal is by piggybacking their communication on other traffic. Catching and editing packets already in transmission helps hide your covert traffic under the authority of whatever service the bot hijacks, as well as ensuring that blocking your bot might block an important service of the user's system.

Authenticity is one of of the trickier security goals to accomplish with running a botnet, as due to the illegal aspect of running a bot, nobody wants a botnet to be traced back to them. To protect the real-life identity of the attacker, it is best to anonymously control the botnet. However, there is always the risk that someone else could discover the botnet. If that were to happen, and that botnet was easily controlled, the person who discovered the attack could hijack it easily by issuing the same commands they see the bots receiving. Because of this, it is important that an APT attacker have some method of verifying the authenticity of each command they send out. An easy way to do this is with public-key cryptography. If the APT attacker has a private key, and all of their bots hold the public key, every command sent can be signed by the attacker and verified by the bot, proving that the command came from the attacker (assuming that the private-key was secure.) This also provides Integrity, as the signature can verify that the message was not altered in transit.

Availability is the security goal of an APT attacker that the attacker has the least control over. Due to the nature of APT attacks, the infrastructure that they run on is not within their control. If someone who actually controls the hardware finds the attack and removes or blocks it, there is not much the attacker can do but attempt to re-infect the host. In order avoid this circumstance, the attacker should be as covert as possible to avoid detection and expulsion from a system. The covert nature of their communication, as talked about above, should help shield the attacker from discovery and removal while still allowing the bot to communicate and remain available. Additionally, the attacker should be as distributed as possible to lessen the uncontrollable na-

ture of their hosts. The distributed nature of the botnet should help mitigate the harm that comes from discovery, as well as the harm that comes from normal computer use. If a user discovers the attack or just shuts down their computer as normal, the botnet should be able to keep moving due to it's wealth of other infected hosts.

An APT attacker has many concerns to keep in mind while running an attack. Most security goals can be achieved with careful work, but it all rests on the covert nature of the attack being used. Discovery often means death for botnets, with patches and defenses being spread through the cybersecurity community quickly. This makes it imperative that the C&C for a botnet is hidden well.

3 Infrastructure

Our infrastructure consists of five VMs and several Python 2.7.2 scripts. Four of these VMs run a version of our bot, which generates cover traffic as well as sending messages using our covert protocol described in the Attack section. The fifth VM is set up as a router, with all traffic on the VM network routing though it. This allows it to act as both a traffic analysis tool for our own testing purposes, as well as a VM to host our defense. Table 1 in the appendix lists each VM's IP and their associated role. As part of the security concerns of this project, we hard-coded each program to work only on the machine with the IP given in the table.

This setup makes valid simplifications, as the scale of the botnet isn't needed to verify that our C&C method is covert. We deploy our traffic analyzer and defense implementation on this router, since both programs need access to the traffic of all bots. Theoretically, it would be more realistic to implement the defense on each VM to block each bot individually. However, this would needlessly complicate the testing, distribution, and logging of the defense.

Each of the bot VMs are set up in almost the exact same way. The VM has three files which compose the bot: bot.py, signfunctions.py, and public_key.pem. The only exception is the first VM, which acts as the "primary" bot. This means it requires the private_key.pem file in order to generate and sign new commands. We chose to have a static "primary" bot for our testing purposes, but an attacker could easily change their IP using a VPN to be less traceable between command injections. Each bot VM also has "NetfilterQueue" installed, allowing us to intercept packets on the kernel level and filter them through our script. The bot.py script is the heart of the bot. It handles the sequencing and splitting of messages, the listening and scraping of packets, and the conversion from message to IPs for our IP Record Route communication. The second file, signfunctions.py, is used to create the signature for each packet with which the bots can verify each command. The ".pem" files were used to store our public and private keys. While these could be combined into one big file, the encapsulation of features helped keep our code clean and our testing simple.

Additionaly, the "primary" bot also had a Python script which could generate, sign, and export commands according to our protocol. We used this system to generate Python object files containing the command we wanted the bot to run. The script was called `makecommand.py` and the commands were saved under `lastcommand.p`.

The router VM was set up using one Python script, in addition with the "NetfilterQueue" and "iptables" commands. The `iptables` command was used on this VM (as well as the four others) to route all traffic on the IP range 10.4.4.0/8 through this VM, and NetfilterQueue was used to intercept this traffic and filter it through our `botbuster.py` script. This forced all communication within the network to flow though this VM, allowing us to log and analyze the traffic using the `tcpdump` command. The `botbuster.py` file is where we implemented all of the packet analysis in our defense. This file can accept or drop packets based on our criteria, preventing them from reaching their destination.

The last part of the process used in this paper is a separate script used to simulate a large network, `infectionsim.py`. This file merely does a simulation of our botnet's communication spread, and can be run on any computer with Python and the graphing library `matplotlib`. This was included to show our C&C method is applicable to a larger botnet, even though our specific implementation focused on a small demonstration network of VMs.

All of the files mentioned within this section, as well as all logs generated during our testing, can be found in the zipped "VM files" folder along with this report.

4 Attack

Our bots utilize the constant stream of UDP packets being sent between players within a match of an online game as cover traffic. Furthermore, our bots hijack these same packets in order to communicate with one another. Specifically, our bots enable IP option 7, which is the IP record route option. This is similar to the method described by Zouheir Trabelsi and Imad Jawhar in their paper "Covert File Transfer Protocol Based on the IP Record Route Option" (Trabelsi and Jawhar 2010). Ordinarily, when this option is set, each router will add its own IP to the list of routers. This can be useful for network diagnostics to see exactly which path a particular packet took through the network. Our implementation prevents this normal functionality, essentially reserving these bytes for ourselves.

In order to use this list of routers for communication, we need to exploit the protocol to make sure this list is

not manipulated by routers on the path. If a particular router believes that the routers list is full, it will not add its own IP to the list. Our attack consists of forging a packet with a "full" list of routers so that it will not be changed as the packet travels through the network. The "pointer" attribute indicates the byte that marks the end of the routers list. By setting the "pointer" attribute to 41, the router will not add its own data to the routers list because the pointer is beyond the 40th byte. In our experimentation, we were able to use 36 bytes of the routers list, and setting the pointer to 41 meant that no additional data was added automatically by our router. Since IP addresses are 4 bytes long, we padded our messages out to be multiples of four bytes. For example, in order to transmit "hello" using this method, the routing record contains the IP addresses 104.101.108.108 and 48.0.0.0 consecutively, where the numbers are the ASCII values of the characters. When decoding a received message, our bots strip off the ending whitespace characters.

The 36-40 bytes used by our covert C&C protocol fits perfectly within the bounds of our theoretical game. "Counter-Strike: Global Offensive" has packet size variation of around 50 bytes (Claypool, LaPoint, and Winslow 2003), allowing our covert messages to blend into normal traffic easily. This can be seen in our "tcpdump-dump.txt" log included with this report.

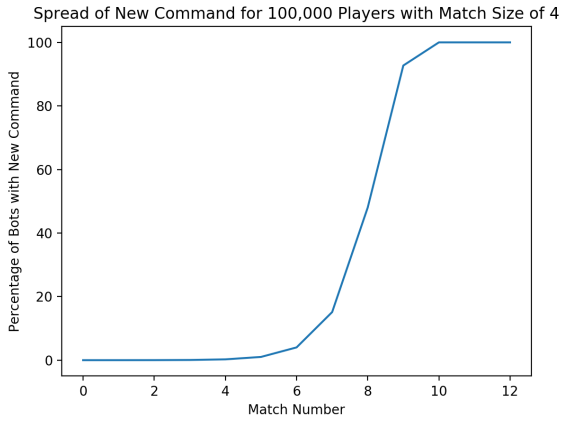
On top of the encoded message, we needed to add two more fields of data to our C&C protocol to obtain all of the functionality we desired: namely, a counter and a cryptographic signature.

The counter was necessary to enable bots to know if the command they were receiving was newer than the previous command they had cached. Commands can take a long time to propagate as players continually match, rematch, and drop out of the games. Because of this, some bots might have an old command they transmit to all other bots within the matches they join. When a bot receives a command, it accepts that command only if the counter is greater, indicating that the command is newer. When a newer command is accepted, they will continue to transmit the new command in both the current match and subsequent matches. The message is signed and verified using the counter and the message contents. This way, the botnet could not be thwarted using an existing command message with an increased counter. We use a counter instead of a timestamp because a timestamp would require each machine to be synchronized, which is difficult without direct control of the hardware.

The cryptographic signature was necessary to ensure that only the attacker could create commands that would be run by the bots. We use SHA to generate 256 byte hashes for the signature, and RSA to generate the keys. The bots, of course, have no knowledge of the secret key used to generate the signature, but can verify the message with the public key, which is known to all bots. This provides us with authenticity, meaning others would not be able to command our botnet through simple replay attacks.

Since a secure signature, plus a counter, plus the command message would not be able to fit within the 36 bytes we were able to utilize per packet, we had to develop a simple file transfer protocol that could reliably transfer a longer string of bytes over the course of a match. Additional rules such as ACKs and NAKs were not necessary to ensure that the command was sent out successfully, since our bots randomly started streaming their most recent command throughout the match. This gave us the redundant retransmission we needed to accommodate for the instances in which a packet was received incorrectly due to the unreliable nature of UDP.

Our networked infrastructure successfully demonstrates our C&C within the scope of a match. In order to prove that our command would be successfully spread to the thousands of computers that would play our theoretical game, we developed an additional simulation. In this simulation, 100,000 players randomly match and rematch other players in games of four. After a match, each bot had the newest command of all bots within that match. For example, if a four player match consists of three bots with a command counter of 100 and one bot with a command counter of 200, the other three bots will have received the new command with a counter of 200. On the other hand, if a match consists of all bots with the same command, no changes are made. The results were promising, demonstrating that the command would reach nearly all bots in around ten matches. If the matches are assumed to be only ten minutes in length, our command reaches all bots in under two hours. The type of growth shown in the results closely resembles the logistic equation, leveling out at the "carrying capacity" of all players in the game. This makes sense intuitively, since the logistic equation can be used as a model for infection (*SI and SIS models*¶ n.d.).



5 Defense

Our defense performs a cascading set of checks on the IP header in order to detect the presence of covert communication. In total, these checks develop a heuristic for "suspiciousness," which we call "sus." We refer to this defense as the "Detecting Uncommon Data Efficiently" defense, or "DUDE" for short. After all of the checks have been completed, DUDE will either accept the packet if "sus" is very low, or drop the packet if "sus" is equal or greater to the "sus threshold." If the packet is "pretty sus" but does not meet the "sus threshold," DUDE may accept the packet anyways, and log the packet for potential future analysis.

In order to defend against covert communication by encoding messages within IP options, we perform four main checks:

The function `s_any_options(spkt)` checks if any option flag is set for the packet. If this is the case, one point is added to "sus." The function `s_ip_options_rr(spkt)` evaluates a packet to see if specifically contains the IP Options Record Route layer. Since there are valid uses for this layer, this is only moderately suspicious when considered on its own. If the packet contains this layer, we simply add 2 points to "sus."

If the previous function returns True, DUDE then continues to evaluate the suspiciousness of the data within the options field with `s_routers_length(spkt)`. We decided that lengthy route records were more suspicious, since they can potentially carry more encoded data. The amount that gets added to "sus" is a function of how many routers have been recorded.

In order to analyze the suspiciousness of the IPs within the route data, one option was to ping all of the routers that are listed in the route data. This is problematic for our particular use case, because in an online video game, many packets may be sent per second. By filling up the route data, an attacker could basically force DUDE to denial-of-service attack all of the routers within its network. Furthermore, if the response from the pings are not immediate, DUDE would have to keep an extensive backlog to simultaneously keep up with all of the incoming packets. Instead, DUDE uses `s_class_array(spkt)` to read through all of the IPs within the route data and assign them to a class. This class can either be A, B, C, D, E, or R. The first four are the main address ranges assigned for IPv4. Class E is a range that is reserved for future use or research (*What is IP (Internet Protocol)?* 2018). We assign the IP as class "R" if it falls into one of the narrow ranges between the classes that are reserved for other purposes. Class E and class R are immediately suspicious, and can be ruled out by DUDE as fake without having to directly check for the existence of that router. Because of this, the amount that is added to "sus" by `s_class_array(spkt)` is a function of the amount of occurrences of class E and class R IPs in the route data. We also considered class D as mildly suspicious due to its rarity. Other strange routings between class A through D could be suspicious, and a future development for DUDE could include another heuristic that takes this into account. However, DUDE can print all of the classes within the route data regardless so that it may be analyzed easily by a network administrator.

These parameters could easily be modified per use case, since the checks DUDE performs are simple calls to Python functions. It is also simple to weight how strongly these checks will affect the "sus" level depending on your use case. For example, if you know that the record route option is commonly used in a particular use case, then you can weight the addition to "sus" in `s_routers_length(spkt)` to be lower, or potentially not perform the check at all. As with the case of Bro, DUDE is most secure when the scripts DUDE uses are kept secret (Paxson 1999). Knowledge of how DUDE calculates "sus" could allow an attacker to build packets that do not add many points to the "sus" level. Since DUDE works on a packet by packet basis, this could be done by spreading the information over more packets.

Our testing showed that these checks were successful in detecting and blocking our botnet. With DUDE running on our simulated networked match, DUDE was able to drop all of the packets that contained covert communication. Since our bot communicates by sending a stream of packets, DUDE could potentially interrupt communication even if only one of the covert packets was caught. This is because missing a packet can only lead to one of two things. Firstly, if the packet that gets dropped is part of the message, the signature will be invalidated by the bot since the message will have changed. Secondly, if part of the signature gets dropped, the message cannot be validated since a complete signature is needed.

Interestingly, the fact that games send UDP packets so rapidly helped hide our C&C messages, but this same fact also drastically decreases the adverse affects of DUDE on gameplay. Since our bots only pack a few legitimate game data packets with the C&C message, most of the packets used for gameplay will get through with DUDE running on the network. The netcode of online games are designed to be robust, and handle dropped packets. In the paper "The Effects of Loss and Latency on User Performance in Unreal Tournament 2003" co-authored by the WPI professor Mark Claypool, the researchers found "Through numerous user studies, we find that packet loss has no measurable affect on player performance in any user interaction category. Moreover, users rarely even notice packet losses even as high as 5% during a typical network game" (Beigbeder et al. 2004). Players might notice brief connectivity issues with DUDE enabled, but connectivity doesn't have to come to a complete halt in order to thwart the botnet.

6 Conclusion

This paper demonstrates the enormous power held by the commander of a botnet. Covert communication protocols are still being discovered. As our communication infrastructure grows in complexity, we run the risk of giving attackers more ways to sneak by. In particular, underdeveloped protocols like those used for games could be rather vulnerable as our networking needs grow.

By hijacking existing game traffic to stream and sign messages, we were able to create a covert protocol that ensures confidentiality, integrity, authenticity, and availability. Authenticity and integrity is provided by the cryptographic signature. Availability was provided by redundant random retransmission throughout matches of the game. Furthermore, the dynamic nature of online games with rapid matchmaking allowed the command to spread to thousands of bots quickly.

Since our C&C protocol can send messages with arbitrary length without creating a significant spike in game traffic, this has serious implications for what similar botnets may be capable of. The commander of a botnet could dynamically update code by sending entire scripts using our covert protocol. Interpreted languages such as Python make this task particularly straightforward. Python can even execute arbitrary strings as code at runtime with the `exec` function. This paper solely aims to explore the mechanics of a covert C&C protocol to implement a system to detect similar botnets; the concerning ramifications of similar bots come as a corollary.

While DUDE was able to detect our command and control protocol, it is an intractable problem to devise a defensive measure that can detect all forms of bots. With knowledge of how DUDE detects "suspiciousness," our own covert C&C protocol could be made even more covert with changes such as encoding messages only as valid, common IP addresses or sending data less frequently. For intrusion detection systems to be continually relevant, it is important for network security experts to anticipate how attackers will attempt to fly under the radar. Since sniffing out creative APTs requires creative countermeasures, the human element of network defense cannot be removed completely. Like DUDE, it is important that these systems cover known threats, and are easily modifiable to adapt to new developments.

References

Antonakakis, Manos et al. (2017). "Understanding the mirai botnet". In: *USENIX Security Symposium*, pp. 1092–1110.

Beigbeder, Tom et al. (2004). "The effects of loss and latency on user performance in unreal tournament 2003®". In: *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*. ACM, pp. 144–151.

Claypool, Mark, David LaPoint, and Josh Winslow (2003). "Network analysis of counter-strike and starcraft". In: *Performance, Computing, and Communications Conference, 2003. Conference Proceedings of the 2003 IEEE International*. IEEE, pp. 261–268.

Paxson, Vern (1999). "Bro: a system for detecting network intruders in real-time". In: *Computer networks* 31.23-24, pp. 2435–2463.

SI and SIS models¶ (n.d.). URL: `https://institutefordiseasemodeling.github.io/Documentation/general/model-si.html`.

Trabelsi, Zouheir and Imad Jawhar (2010). "Covert file transfer protocol based on the IP record route option". In: *Journal of Information Assurance and Security* 5.1, pp. 64–73.

What is IP (Internet Protocol)? (2018). URL: `https://www.computerhope.com/jargon/i/ip.htm`.

7 Appendix

VM IP	Role in infrastructure
10.4.4.1	The primary bot, used to issue the "newest" commands.
10.4.4.2	Used as "Router" to sniff traffic for testing, as well as a simulated firewall.
10.4.4.3	Unused
10.4.4.4	Another Bot
10.4.4.5	Another Bot
10.4.4.6	Another Bot

Table 1: Roles of each VM