Social Zombies: Aspects of Trojan Networks

warlord

warlord / nologin.org

Contents

Chapter 1

Introduction

While I'm sitting here and writing this article, my firewall is getting hammered by lots and lots of packets that I never asked for. How come? In the last couple of years we saw the internet grow into a dangerous place for the uninitiated, with worms and viruses looming almost everywhere, often times infecting systems without user interaction. This article will focus on the subclass of malware commonly referred to as worms, and introduce some new ideas to the concept of worm networks.

Chapter 2

Worm Infection Vectors

The worms around today can mostly be put into one the four categories discussed in the following sections.

2.1 Mail

The *mail* worm is the simplest type of worm. It's primary mode of propagation is through social engineering. By sending large quantities of mail with content that deceives people and/or triggers their curiosity, recipients are tricked into running an attached program. Once executed, the program will send out copies of itself via email to recipients found in the victims address book. This type of worm is usually stopped quickly when antivirus companies update their signature files, and mail servers running those AV products start filtering the worm mails out. Users, in general, are becoming more and more aware of this type of malware, and many won't run attachments sent in mail anymore. Regardless, this method of infection still manages to be successful.

2.2 Browser

Browser-based worms, which primarily work against Internet Explorer, make use of vulnerabilities that exist in web-browsers. What generally happens is that when a users visits a malicious website, an exploit will make Internet Explorer download and execute code. As there are well known vulnerabilities in Internet Explorer at all times that are not yet fixed, the bad guys usually have a couple of days or weeks to spread their code. Of course, the infection rate heavily depends on the number of visitors on the website hosting the exploit. One approach that has been used in the past to gain access to a wider 'audience' involved sending mail to thousands of users in an attempt to get the users to visit a malicious website. Another approach involved hacking advertisement companies and changing their content in order to make them serve exploits and malware on high profile sites.

2.3 Peer to Peer

The peer to peer worm is quite similar to the mail worm; it's all about social engineering. Users hunting for the latest mp3s or pictures of their most beloved celebrity find similarly named programs and scripts, trying to deceive the user to download and execute them. Once active on the users system, the malcode will make sure it's being hosted by the users p2p application to spread further. Even if downloaded, host based anti-virus scanners with recent signatures will catch most of these programs before they can be run.

2.4 Active

This one is the most dangerous worm, as it doesn't require any sort of user interaction at all. It also requires the highest level of skill to write. Active worms spread by scanning the internet for one or more types of vulnerabilities. Once a vulnerable target is found, an exploit attempt is made that, if successful, results in the uploading of the worm to the attacked site where propagation can continue in the same form. These worms are usually spotted first by an increasing number of hosts scanning the internet, most often scanning for a single port. These worms also usually exploit weaknesses that are well-known to the public for hours, days, weeks or months. Examples of this type of worm include the Wank worm, Code Red, Sadmind, SQL Slammer, Blaster, Sasser and others. As the use of firewalls and NAT routers increases, and as anti-exploit techniques like the one employed by Windows XP SP2 become more common, these worms will find less hosts to infect. To this point, from the time of this writing, it's been a while since the last big active worm hit the net.

Other active infection vectors include code spreading via unset or weak passwords on CIFS[1] shares, IRC and instant messaging networks, Usenet, and virtually every other data exchange protocol.

[1]Common Internet File System. The protocol used to exchange data between Windows hosts via network shares.

Chapter 3

Motives

3.1 Ego

Media attention often is a major motivation behind a worm. Coders bolstering their ego by seeing reports on their worm on major sites on the internet as well as tv news and newspapers with paniced warnings of the latest doomsday threat which may take down the planet and result in a 'Digital Pearl Harbor' $^{\text{TM}}$seems to be quite often the case. Huge media attention usually also means huge law enforcement attention, and big efforts will be made to catch the perpetrator. Though especially wide open (public) WIFI networks can make it quite difficult to catch the perpetrator by technological means, people boasting on IRC and, as in the case of Sasser, bounties, can quickly result in the worm's author being taken into custody.

3.2 DDoS

The reason for a DDoS botnet is usually either the wish to have enough firepower to virtually shoot people/sites/organizations off the net, or extortion, or a combination of both. The extortion of gambling websites before big sports events is just one example of many cases of extortion involving DDoS. The attacker usually takes the website down for a couple of hours to demonstrate his ability to do so whenever it pleases him, and sends a mail to the owner of the website, asking for money to keep the firepower away from his site. This sort of business model is well known for millenia, and merely found new applications online.

3.3 Spamming

This one is also about money in the end. Infected machines are (ab)used as spam zombies. Each machine sends their master's unsolicited mail to lots and lots of unwilling recipients. The owners of these systems usually offer their services to the spam industry and thus make money of it.

3.4 Adware

Yet another reason involving money. Just like on TV and Google, advertisements can be sold. The more people seeing the advertisement, the more money can be requested from the people that pay for their slogan to be displayed on some end users Windows. (Of course, it could be Linux and MacOS too, but, face it, no adware attacks those)

3.5 Hacking

A worm that infects and backdoors a couple thousand hosts is a great way to quickly and easily obtain data from those systems. Examples of data that may be worth stealing includes accounts for online games, credit card numbers, personal information that can be used in identity theft scams, and more. There has even been a report that items of online games were being stolen to sell those later on E-bay. Already having compromised one machine, enhancing the influence into some network can be much easier of course. Take for example the case of a heavily firewalled company. A hacker can't get inside using an active approach, but notices that one of his malware serving websites infected a host within that network. Using a connect-back approach, where the infected node connects to the attacker, a can tunnel can be built through the firewall thereby allowing the attacker to reach the internal network.

Chapter 4

Botnets

While I did mention DDoS and spam as reasons for infection already, what I left out so far was the infrastructure of hundreds or thousands of compromised machines, which is usually called a `botnet`. Once a worm has infected lots of systems, an attacker needs some way to control his zombies. Most often the nodes are made to connect to an IRC server and join a (password protected) secret channel. Depending on the malware in use, the attacker can usually command single or all nodes sitting on the channel to, for example, DDoS a host into oblivion, look for game CD keys and dump those into the channel, install additional software on the infected machines, or do a whole lot of other operations. While such an approach may be quite effective, it has several shortcomings.

1. IRC is a plaintext protocol.

   Unless every node builds an SSL tunnel to an SSL-capable IRCD, everything that goes on in the channel will be sent from the IRCD to all nodes connected, which means that someone sniffing from an infected honeypot can see everything going on in the channel, including commands and passwords to control the botnet. Such a weakness allows botnets to be stolen or destroyed (f.ex. by issuing a command to make them connect to a new IRCD which is on IP 127.0.0.1).

2. It's a single point of failure.

   What if the IRCD goes down because some victim contacted the admin of the IRC server? On top of this, an IRC Op (a IRC administrator) could render the channel inaccessible. If an attacker is left without a way to communicate with all of the zombie hosts, they become useless.

A way around this dilemma is to make use of dynamic DNS sites like www.dyndns.org. Instead of making the zombies connect to irc.somehost.com, the attacker can install a dyndns client which then allows drones to reference a hostname that can be directed to a new address by the attacker. This allows the attacker to migrate zombies from one IRC server to the next without issue. Though this solves the problem of reliability, IRC should not be considered secure enough to operate a botnet successfully.

The question, then, is what is a better solution? It seems the author of the trojan `Phatbot` already tried to find a way around this problem. His approach was to include peer to peer functionality in his code. He ripped the code of the P2P project "Waste" and incorporated it into his creation. The problem was, though, that Waste itself didn't include an easy way to exchange cryptographic keys that are required to successfully operate the network, and, as such, neither did Phatbot. The author is not aware of any case where Phatbot's P2P functionality was actually used. Then again, considering people won't run around telling everyone about it (well, not all of them at least), it's possible that such a case is just not publicly known.

To keep a botnet up and running, it requires reliability, authentication, secrecy, encryption and scalability. How can all of those goals be achieved? What would the basic functionality of a perfect botnet require? Consider the following points:

1. An easy way to quickly send commands to all nodes
2. Untraceability of the source IP address of a command
3. Impossibile to judge from an intercepted command packet which node it was addressed to
4. Authentication schemes to make sure only authorized personnel operate the zombie network
5. Encryption to conceal communication
6. Safe software upgrade mechanisms to allow for functionality enhancements
7. Containment; so that a single compromised node doesn't endanger the entire network
8. Reliability; to make sure the network is still up and running when most of its nodes have gone
9. Stealthiness on the infected host as well as on the network

At this point one should distinguish between `unlinked` and `linked`, or passive, botnets. Unlinked means each node is on its own. The nodes poll some central resource for information. Information can include commands to download software updates, to execute a program at a certain time, or the order a DDoS on a given target machine. A linked botnet means the nodes don't do anything by themselves but wait for command packets instead. Both approaches have advantages and disadvantages. While a linked botnet can react faster and may be more stealthy considering the fact that it doesn't build up periodic network connections to look out for commands, it also won't work for infected nodes sitting behind firewalls. Those nodes may be able to reach a website to look for commands, which means an unlinked approach would work for them, but command packets like in the linked approach won't reach them, as the firewall will filter those out. Also, consider the case of trying to build up a botnet with the next Windows worm. Infected Windows machines are generally home users with dynamic IP addresses. End-user machines change IPs regularly or are turned off because the owner is at work or on a hunting weekend. Good luck trying to keep an up-to-date list of infected IPs. So basically, depending on the purpose of the botnet, one needs to decide which approach to use. A combination of both might be best. The nodes could, for example, poll a resource of information once a day, where commands that don't require immediate attention are waiting for them. On the other hand if there's something urgent, sending command packets to certain nodes could still be an option. Imagine a sort of unlinked botnet. No node knows about another node and nor does it ever contact one of its brothers, which perfectly achieves our goal of containment. These nodes periodically contact what the author has labeled a `resource of information` to retrieve their latest orders. What could such a resource look like?

The following attributes are desirable:

1. It shouldn't be a single point for failure, like a single host that makes the whole system break down once it's removed.
2. It should be highly anonymous, meaning connecting there shouldn't be suspicious activity. To the contrary, the more people requesting information from it the better. This way the nodes' connections would vanish in the masses.
3. The system shouldn't be owned by the botnet master. Anonymity is one of the botnet's primary goals after all.
4. It should be easy to post messages there, so that commands to the botnet can be sent easily.

There are several options to achieve these goals. It could be:

1. Usenet: Messages posted to a large newsgroup which contain steganographically hidden commands that are cryptographically signed achieves all of the above mentioned goals.

2. P2P networks: The nodes link to a server once in a while and, like hundreds of thousands of other people, search for a certain term ("xxx"), and find command files. File size could be an indicator for the nodes that a certain file may be a command file.
3. The Web itself: This one would potentially be slow, but of course it's also possible to setup a website that includes commands, and register that site with a search engine. To find said site, the zombies would connect to the search engine and submit a keyword. A special title of the website would make it possible to identify the right page between thousands of other hits on the keyword, without visiting each of them.

Using those methods, it would be possible to administer even large botnets without even having to know the IP adresses of the nodes. The "distance" between botnet owner and botnet drone would be as large as possible since there would be no direct connection between the two. These approaches also face several problems, though:

How would the botnet master determine the number of infected hosts that are up and running? Only in the case of the website would estimation of the number of nodes be possible by inspecting the access logs, even logging were to be enabled. In the case of the Usenet approach a command of "DDoS Ebay/Yahoo/Amazon/CNN" might just reach the last 5 remaining hosts, and the attacker would only be left with the knowledge that it somehow didn't work. The problem is, however, that the attacker would not know the number of zombies that would actually take part in the attack. The same problem occurs with the type and location of the infected hosts. Some might be high profile, such as those connecting from big corporations, game developers, or financial institutions. The attacker might be interested in abusing those for something other than Spam and DDoS, if he knew about them in particular. If the attacker wants to bounce his connections over 5 of his compromised nodes to make sure he can't be traced, then it is required that he be able to communicate with 5 nodes only and that he must know address information about the nodes. If the attacker doesn't have a clue which IP addresses his nodes have, how can he tell 5 of them where to connect to? Besides the obvious problem of timing, of course. If the nodes poll for a new command file once every 24 hours, he'd have to wait 24 hours in the worst case until the last node finds out it's supposed to bind a port and forward the connection to somewhere else.

4.1 The Linked Network

Though I called this approach a passive network, as the nodes idle and wait for commands to come to them, this type of botnet is in fact quite active. The mechanisms described now will not (easily) work when most of the nodes are on dynamic IP addresses. It is thus more interesting for nodes installed after exploiting some kind of server software. Of course, while not solving the uptime problem, a rogue dyndns account can always give a dynamic IP a static hostname.

This kind of network focuses on all of its nodes forming some kind of self-organizing peer to peer network. A node that infects some other host can send over the botnet program and make the new host link to itself, thus becoming that node's parent. This technique can make the infected hosts form a sort of tree structure over time, as each newly infected host tries to link to the infecting host. Updates, information, and commands can be transmitted using this worm network to reach each node, no matter which node it was sent from, as each node informs both child nodes as well as its parent nodes. In its early (or final) stages, a network of this type might look like this piece of ascii art:

```
Level
0      N
     /   \
1   N     N
   / \   /
2 N   N N
```

To make sure a 'successful' node that infects lots of hosts doesn't become the parent of all of those hosts, nodes must refuse link requests from child nodes after a certain number have been linked (say 5). The parent can instead in form the would-be child to link to one of its already established children instead. By keeping track of the number of nodes linked to each location in the tree, a parent can even try to keep the tree thats hierarchically below it well balanced. This way a certain node would know about its parent and up to 5 children, thus keeping the number of other hosts that someone who compromises a node rather low, while still making sure to have a network that's as effective as possible. Depending on the number of nodes in the entire network, the amount of children that may link to a parent node could be easily changed to make the network scale better. As each node may be some final link as well as a parent node, every host runs the same program. There's no need for special 'client' and 'server' nodes.

Whats the problem with a tree structure? Well, what if a parent fails? Say a node has 3 children, each having 2 children of its own. Now this node fails because the owner decides to reinstall the host. Are we left with 3 networks that can't communicate with each other any more? Not necessarily. While possibly giving a forensics expert information on additional hosts, to increase reliability each node has to know about at least one more upstream node that it can try to link to if its parent is gone. An ideal candidate could be the parent's parent. In order to make sure that all nodes are still linked to the network, a periodic (once a day) sort of "ping" through the entire network has to happen in any case. By giving a child node the IP of its "grandparent", the direct parent of the child node always knows that the fail-over node, the one its kids will try to link to if it should fail, is still up and running.

Though this may help to address the issue of parent death, another issue remains. If the topmost node fails, there are no more upstream nodes that the children could link to. Thats why in this case the children should have the ip of one(!) of its siblings as the fail-over address so that they can make this one the new top node in the case of a fail-over condition. Making use of the node-based ping, each node also knows how many of its children are still up and running. By including this number into the ping sent to the parent, the topmost node could always tell the number of linked hosts. In order to not have to rely on connecting to the topmost node to collect this type of information, a simple command can be implemented to make the topmost node report this info to any node on the network that asks for it. Using a public key stored into all the nodes, it's even possible to encrypt every piece of information thats destined for the botnet owner, making sure that no one besides the owner can decrypt the data. Although this type of botnet may give a forensics expert or someone with a sniffer information on other nodes that are part of the network, it also offers fast response times and more flexibility in the (ab)use of the network compared to the previous approach with the unlinked nodes. It's a sort of trade off between the biggest possible level of anonymity on one hand, and flexibility on the other. It is a huge step up compared to all of the zombies sitting on IRC servers right now, where a single channel contains the zombies of the entire botnet. By employing cryptography to store the IPs of the child and parent nodes, and keeping those IPs only in RAM mitigates the problem further.

Once a drone network of this type has been established with several hundreds of hosts, there are lots of possibilities of putting it to use. To conceal the originating IP address of a connection, hopping over several nodes of the drone network to a target host can be easily accomplished. A command packet tells one node to bind a port. Once it receives a connection on it, it is told to command a second node to do the same, and from then on node 1 forwards all the traffic to node 2. Node 2 does the same, and forwards to node 3, then 4, maybe 5, until finally the last node connects to the intended destination IP. By encrypting the entire connection from the original source IP address up to the last node, a possible investigator sniffing node 2 will not see the commands (and thus the IP addresses) which tell node 3 to connect to node 4, node 4 to node 5, and of course especially not the destination host's address. An idle timeout makes sure that connections don't stay up forever.

As manually updating several hundreds or thousands of hosts is tedious work, an easy updating system should be coded into the nodes. There are basically two possible ways to realize that. A command, distributed from node to node all over the network, could make each node replace itself with a newer version which it may download from a certain HTTP address. The other way is by updating the server software on one node, which in turn distributes this update to all the nodes it's linked to (children *and* parent), which do just the same. Cryptographic signatures are a must of course to make sure someone doesn't replace all of the precious nodes with SETI@home. Vlad902 suggested a simple and effective way to do that. Each node gets an MD5 hash hardcoded into it. Whenever someone offers a software update, it will download the first X bytes and see wether they hash to the hardcoded value. If they do, the update will be installed. Of course, a forensics expert may extract the hash out of an identified node. However, due to the nature of cryptographic hashes, he won't be able to tell which byte sequence generates that hash. This will prevent the forensics export from creating a malicious update to take down the network. As the value used to generate the hash has to be considered compromised after an update, each update has to supply a new hash value to look out for.

Further security mechanisms could include making the network completely memory resident, and parents keeping track of kids, and reinfecting those if necessary. What never hit the hard-disk can obviously not be found by forensics. Also, commands should be time-stamped to make sure a certain command will only work once, and replay attacks (sending a sniffed command packet to trigger a response from a node) will fail. Using public key cryptography to sign and encrypt data and communication is always a nice idea too, but it also has 2 disadvantages:

1. It usually produces quite a big overhead to incorporate into the code.
2. Holding the one and only private key matching to a public key thats been found on hundreds of hacked hosts is quite incriminating evidence.

An additional feature could be the incorporation of global unique identifiers into the network, providing each node with a unique ID that's set upon installation on each new victim. While the network master would have to keep track of host addresses and unique IDs, he could use this feature to his advantage. Imagine a sort of traceroute within the node network. The master wants to know where a certain host is linked to. Every node knows the IDs of all of the child nodes linked hierarchically below it. So he asks the topmost node to find out the path to the node he's interested in. The topmost node realizes it's linked somewhere under child 2, and in turn asks child 2. This node knows it's linked somewhere below child 4, and so on and so on. In the end, the master gets his information, a couple of IDs, while no node thats not directly linked to another gets to know the IPs of further hosts that are linked to the network.

Since a portscan shouldn't reveal a compromised host, a raw socket must be used to sniff command packets off the wire. Also, command packets should be structured as unsuspicious as possible, to make it look like the host just got hit by yet another packet of "internet background noise". DNS replies or certain values in TCP SYN packets could do the trick.

4.2 The Hybrid

There is a way to combine both the anonymity of an unlinked network with the quick response time of the linked approach. This can be done by employing a technique first envisioned in the description of a so-called “Warhol Worm”. While no node knows anything about other nodes, the network master keeps track of the IPs of infected hosts. To distribute a command to a couple or maybe all of the nodes, he first of all prepares an encrypted file containing the IPs of all active nodes, and combines that with the command to execute. He then sends this commandfile to the first node on the list. This node executes the command, takes itself from the list, and goes top to bottom through the list, until it finds another active node, which it transmits the command file to. This way each node will only get to know about other nodes when receiving commandfiles, which are subsequently erased after the file has been successfully transmitted to another node. By calling certain nodes by their unique IDs, it’s even possible to make certain nodes take different actions than all the others. By preparing different files and sending them to different nodes at start already, quite a fast distribution time can be achieved. Of course, should someone accomplish to not only sniff the commandfile, but also decrypt it, he has an entire list of infected hosts. Someone sniffing a node will still also see an incoming connection from somewhere, and an outgoing connection to somewhere else, and thus get to know about 2 more nodes. Thats just the same as depicted in the passive approach. Whats different is that a binary analysis of a node will not divulge information on another host of the network. As sniffing is probably more of a threat than binary analysis though, and considering a linked network offers way more flexibility, the Hybrid is most likely an inferior approach.

Chapter 5

Conclusion

When it comes to botnets, the malcode development is still in it's infancy, and while today's networks are very basic and easily detected, the reader should by now have realized that there are far better and stealthier ways to link compromised hosts into a network. And who knows, maybe one or more advanced networks are already in use nowadays, and even though some of their nodes have been spotted and removed already, the network itself has just not been identified as being one yet.

Bibliography

[1] The Honeypot Project. *Know Your Enemy: Tracking Botnets.*
http://www.honeynet.org/papers/bots/

[2] Weaver, Nicholas C. *Warhol Worms: The Potential for Very Fast Internet Plagues.*
http://www.cs.berkeley.edu/~nweaver/warhol.html

[3] Paxson, Vern, Stuart Staniford & Nicholas Weaver. *How to 0wn the Internet in Your Spare Time.*
http://www.icir.org/vern/papers/cdc-usenix-sec02/

[4] Zalewski, Michael. *Writing Internet Worms for Fun and Profit.*
http://www.securitymap.net/sdm/docs/virus/worm.txt