Overbot - A botnet protocol based on Kademlia

Guenther Starnberger
Distributed Systems Group
Vienna Univ. of Technology
gst@infosys.tuwien.ac.at

Christopher Kruegel
UC Santa Barbara
chris@cs.ucsb.edu

Engin Kirda
Eurecom
kirda@eurecom.fr

ABSTRACT

One crucial point in the implementation of botnets is the command and control channel, which is used by botmasters to distribute commands to compromised machines and to obtain results from previous commands. While the first botnets were mainly controlled by central IRC servers, recent developments have shown the advantages of a more decentralized approach using peer-to-peer (P2P) networks. Interestingly, even though some botnets already use P2P networks, they do so in a naive fashion. As a result, most existing botnet implementations allow attackers to disrupt messages from the botmaster and to learn IP addresses of other nodes within the botnet.

This paper introduces Overbot, a botnet communication protocol based on a peer-to-peer architecture. More precisely, Overbot leverages Kademlia, an existing P2P protocol, to implement a stealth command and control channel. An attacker can neither learn the IP addresses of other nodes in the botnet nor disrupt the message exchange between the botmaster and the bots, even when the attacker is able to capture some of the nodes within the network. Overbot demonstrates the threats that may result when future botnet generations utilize more advanced communication structures. We believe that it is important to outline these threats to allow the research community to develop solutions before such botnets appear in the wild. To help the search for effective countermeasures, we also discuss possible directions where future research seems promising.

Categories and Subject Descriptors

C.2.0 [**Computer Systems Organization**]: Communication/Networking and Information Technology—*Network-level security and protection*

Keywords

Botnet Protocol, Malware, Network Security

SecureComm 2008 September 22 - 25, 2008, Istanbul, Turkey

1. INTRODUCTION

A communication channel in a botnet allows the botmaster to issue commands to the nodes within the network and to receive replies from the individual nodes. Many older botnets used the Internet Relay Chat (IRC) protocol as a means of communication [5]. This makes them vulnerable to several kinds of attacks. First, the operator of the IRC service is able to block the channel that is used for communication. Furthermore, it is possible to obtain the IP addresses of hosts that join an IRC channel that is used by a botnet. Additionally, the list of IRC servers and IRC channels used for botnet communication may be distributed as a blacklist. An advanced firewall may be able to drop connections when a host tries to join a channel that is used by botnets.

Over time, botmasters have recognized the weaknesses of a command and control channel that is based on a centralized IRC server. To address the problems, new botnet variations emerged that follow a distributed communication approach [3]. In particular, because legitimate peer-to-peer systems are already wide-spread, it is straightforward to abuse these existing applications. An example of a P2P-based botnet and, so far, the most successful representative is Trojan.Peacomm [5], also known as Storm. Peacomm uses the distributed hash table of a P2P file-sharing system for its communication, hiding its activity within legitimate application traffic.

An innovative feature of Peacomm is that the botmaster does not have to contact bots directly. Instead, the botmaster stores commands under specific search keys, which are derived from the current date. The bots independently calculate the current search key and request these keys to obtain the commands. Because the keys change frequently, new commands can be injected easily. Of course, an attacker[1] can capture bot instances and identify the keys that Peacomm is searching for. This allows the attacker to join the network with a node ID close to that key. When requests for this key are received, one can collect the IP addresses of nodes that are part of the botnet.

Other advanced botnets [13, 14] build custom P2P networks and attempt to limit the information that an attacker can obtain by capturing a single node. The common approach is to let single nodes only know about few other nodes. This is achieved by splitting a large botnet into sev-

[1]Throughout this paper, we use the word "attacker" to refer to a white-hat security person that attempts to disrupt and detect botnets. We call this person attacker because we assume the role of the botmaster here.

eral smaller botnets, restricting the connections of each node to a small subset of other, compromised hosts.

In all the aforementioned approaches, an attacker is able to learn some information about the botnet after capturing a node. Because attackers can often control a significant number of nodes, large fractions of the botnet can be revealed.

This paper presents the botnet communication protocol Overbot. It shows that it is possible to create a botnet where a captured node reveals no information about the other nodes in the botnet. Its design assures that:

- An attacker who is able to capture some of the nodes is not able to identify any of the remaining nodes.
- The behavior of nodes within the botnet is not distinguishable from legitimate P2P nodes.
- The botmaster does not contact the botnet nodes directly. Therefore, botnet nodes do not obtain the IP addresses that are used by the botmaster to issue commands.

Experiments conducted on the Bittorrent distributed hash table (DHT) implementation show the feasibility of our approach.

We believe that it is important for the security community to study the design space of botnet command and control structures. The reason is that current techniques that are effective against today's botnets may not be sufficient to disrupt future botnet generations. Overbot serves as an example of a certain type of future botnet. This allows researches to develop techniques against these kinds of botnets before they appear in the wild. Furthermore, new P2P protocols should be designed in a way that makes it harder for bots to hide between legitimate nodes.

2. INTRODUCTION TO KADEMLIA

Overbot is based on Kademlia [7], a distributed hash table (DHT) used by several P2P applications. There exist slightly different implementations of the Kademlia protocol that are used on the Internet. One popular variant is used by eMule, another one by Bittorrent. These differences require small adaptations to our Overbot protocol. In the following paragraphs, we will first present the underlying, general protocol. Then, we will discuss the specifics of two popular variants (Kad and Kashmir) that are relevant for Overbot.

In Kademlia, nodes are identified by 160-bit node IDs, and data items are identified by 160-bit keys. A node usually stores data items whose key values are close to its own node ID. Key values of data items are typically calculated by using a hash function, while node IDs are randomly generated by the nodes themselves. The distance between two values is defined by the XOR metric $d(x, y) = x \oplus y$, where x and y are the node IDs of the respective nodes. As the distance is only based on IDs, a low distance does not implicate any kind of closeness on the underlying network.

As a routing table, Kademlia maintains lists that contain information about nodes in the distance from 2^i to 2^{i+1}, where $0 \leq i < 160$. These lists are termed as k-buckets. Nodes in the k-buckets are sorted by the time when the least recently sent message of a particular node was received and each k-bucket can store up to k elements.

Generally, k-buckets prefer old nodes to new nodes. If a message is received from a new node and the k-bucket is not full, information about this node is stored in the k-bucket. If the k-bucket is full, it is checked whether the least-recently seen node responds to a `PING` request. If it responds, information about the new node is not stored, otherwise the stored information about the least-recently seen node is replaced by the information of the new node. Therefore, an old node is only evicted from the k-bucket if no recent messages have been received and if it fails to respond to `PING` requests.

The Kademlia protocol defines four RPC (remote procedure call) type messages: `PING`, `STORE`, `FIND_NODE` and `FIND_VALUE`.

Nodes are located iteratively by using the `FIND_NODE` RPC function. When a node wants to find another node, it issues a `FIND_NODE` request to k of its neighbors, providing the ID of that node as argument. Each neighbor will then return a list of its closest k neighbors. The original node stores these new node IDs and continues querying nodes, until no better results are returned or until the desired node ID is found.

`FIND_VALUE` works similarly to `FIND_NODE`. Each nodes returns its closest k neighbors. However, if a node contains the requested key, it returns the data item associated with the key instead of a list of neighbors. After the original requester received this item, it stores the data at the neighbor node nearest to the key that did not return the value.

`STORE` is used to store data in the DHT. By using `FIND_NODE`, a node first determines k nodes that are located close to the key value. Afterwards, it sends `STORE` requests to all of these nodes. In order to ensure persistence of stored values, each node republishes each value that it stores once per hour to the k nodes closest to the particular value.

The `PING` RPC allows a node to check whether another node is online. Furthermore, a `PING` reply also contains the respective node ID.

In order to initially connect to the network, the IP address of at least one node in the DHT is required. This node can then be queried about other nodes close to the own node ID. To this end, clients typically contain a list of hard-coded nodes (IP addresses) that serve as initial contact points.

2.1 Kad

Kad [8] is an implementation of the Kademlia DHT that is used by several P2P applications. Examples of applications that use Kad are Overnet and eMule.

When a P2P application wants to publish information about a file in the Kad network, it first creates a MD4 hash of this file. This MD4 hash is then used as the key of a *location entry*. Such a location entry contains the node ID as well as the IP address and port number of the node on which the file is available. To make it possible to search for a file by its name, each word in the filename is first converted to lowercase and then hashed to a MD4 value. Each of these values is used as the key of a DHT record that contains the MD4 hash of the file, together with additional meta information about the file such as the type or the file size.

An example of an entry that contains meta information about a file is: `#1234 #5678 NAME=test.avi;SIZE=123456; TYPE=Video`, where `#1234` is the key of the entry and the hash value of a term in the filename, and `#5678` is the MD4 hash of data inside the file and the key of a location entry. An example of such a location entry is: `#5678 #9876`

`loc=bcp://192.168.3.10:4661`, where `#5678` is the MD4 value of the data and `#9876` is the node ID of the host containing the data. The `loc` entry indicates how the host in question can be contacted.

Assume that a client wants to search for the file `test.avi`. The client first hashes the filename to `#1234`. The first search result yields a DHT entry with `#5678` as value. This value is used as key for the second search, this time to find the location entry that stores the file location.

2.2 Khashmir

Khashmir [6] is another Kademlia implementation, which is used in the Bittorrent protocol. Bittorrent uses the DHT to allow clients to find other peers without requiring a central tracker. Each Bittorrent download is identified by a 160-bit value called an *infohash*. Under the DHT key of the infohash, the IP addresses and port numbers of all peers currently downloading the file in question are stored.

The Khashmir protocol is based on a RPC mechanism that allows peers to call remote procedures of other peers. This mechanism is used by the Bittorrent protocol to implement the `announce_peer` and `get_peers` functionalities. If a peer starts a download, it calls `announce_peer` at other nodes close to the ID of the infohash. This notifies the peers that the node is downloading the file. Other peers can use the `get_peers` RPC to retrieve a set of nodes that are currently downloading a specific file. `get_peers` is comparable to the `FIND_VALUE` function in eMule.

Unlike eMule, the DHT implementation of Bittorrent *restricts* the information stored in the DHT to IP addresses and port numbers. It is not possible to store any other information. This is important because it restricts the freedom for Overbot to store information in those entries.

3. THREAT MODEL

3.1 Attacks against botnets

There are different levels at which a botnet and the compromised nodes that are part of this botnet can be attacked:

- Directly at the host on which the bot is running. For example, an attacker might capture and analyze one of the nodes that are part of the botnet.
- At the network level, when an attacker has access to the traffic that is sent and received by botnet nodes.
- At the application level, when the botnet makes use of a legitimate protocol (such as IRC or P2P file-sharing), and the attacker joins the system and monitors the node's behavior at the application-level.

Attacks at these three levels are discussed in more detail in the following paragraphs.

Host level.

We assume that an attacker is able to capture an arbitrary number of botnet nodes, and he can completely reverse engineer the software running on these nodes. All information stored on a captured node is, therefore, accessible to the attacker. If a node is able to identify other nodes or the botmaster, this knowledge is also available to the attacker.

Network level.

At the network level, we assume that an attacker has full access to the traffic sent and received by a number of botnet and legitimate nodes. An example for such an attacker is a firewall or an ISP. An attacker can analyze the traffic generated by a node to determine whether this host is part of the botnet. Thus, the behavior exhibited by a compromised node must not obviously differ from any legitimate DHT client. In particular, this is a problem for botnet designs that make use of custom protocols that "stand out" in normal network traffic. To a lesser degree, this is also true for statistical patterns. If a node shows abnormal behavior, this might reveal its botnet membership.

Application level.

An attacker who is able to join the application that is abused by a botnet has several possibilities to identify compromised nodes. These possibilities are often related to observable differences between the behavior of a bot and a legitimate node. An attacker might observe that a node is always querying for non-existent keys in a distributed hash table. Or a bot might issue a suspiciously large number of requests. For example, in Kademlia, it may be unlikely that a legitimate node issues a lot of `FIND_VALUE` requests within a short time-span.

Often, an attacker can also launch a Sybil attack [4] to obtain control of a large set of nodes that are part of the botnet. This does not only give the attacker a higher chance to identify suspicious queries, but he might also try to detect statistical patterns that are typical for a member of the botnet.

Combined attacks.

Clearly, information obtained at different levels can be combined to launch a successful attack. An example of an attack that leverages combined knowledge from the host and the network level is the way in which Trojan.Peacomm nodes can be identified. First, the attacker captures a compromised node to predict keys that bots are searching for to obtain commands. Then, the attacker joins the P2P network and injects nodes that have IDs close to the search key. Finally, the attacker waits for other bots to request this key. At this point, the attacker can learn the IP addresses of the malicious nodes.

3.2 Weak points in current botnets

We note that Overbot is designed to withstand attacks at all three levels. This is not true for current botnet protocols; a fact that allows detection of nodes and disruption of communication in these systems. Details of the shortcomings of current botnet designs are outlined below.

In IRC-based botnets, capturing a node reveals the name of the IRC network and the name of the IRC server to the attacker. If the traffic to the IRC server is not encrypted, it can also be used to obtain the name of the IRC channel. On many IRC networks, any member of a channel is able to obtain the IP addresses of other members. If this feature is not available to normal users, the cooperation of the administrator of the IRC service is required. In both cases, the IPs of the clients that are logged on can be used to identify compromised hosts. IRC-based approaches are, therefore, vulnerable to host and network level attacks.

As mentioned previously, the shortcomings of IRC-based command and control structures prompted botmasters to seek alternatives that rely on more robust, distributed designs. A number of botnets, which use more advanced network designs, are discussed in [5]: Among the notable ones are Phatbot and Nugache. Phatbot uses the Gnutella P2P network; infected nodes can identify other nodes because they use a fixed version string, together with a non-standard, fixed port-number [12]. Such behavior is easy to identify at the network level. In Nugache, the binary is equipped with a list of 22 server IPs to which an infected host can connect. When connecting, the servers send an updated server list to the bot. Therefore, filtering access to these 22 hard-coded IPs prevents bots from updating their server list and, as a result, from receiving commands. Additionally, Nugache listens on port number 8; this makes it straightforward to identify any Nugache-related traffic. A number of additional possibilities to detect the aforementioned botnets are described in [11].

Trojan.Peacomm [5] uses a botnet protocol based on the Overnet flavor of the Kademlia DHT. In Peacomm, both, botmaster and nodes calculate a set of hash values using a function that takes the current date as an input. The botmaster can store commands to the nodes under these hash values, and the nodes fetch their content in regular intervals. To receive messages from the botmaster, nodes need to actively request the content of specific hash values. As mentioned previously, Trojan.Peacomm uses predictable DHT hash values under which commands to the bots are stored. At the network level, an attacker can identify DHT search requests for these hash values, detecting infected hosts. Moreover, at the application level, an attacker may assign his nodes an ID close to one of the hash values used by Trojan.Peacomm. Therefore, many bots will issue their search requests to the attacker. This allows the attacker to obtain a list of IP addresses on which botnet nodes may be active.

The approach explored in the paper "Army of Botnets" [13] splits a large botnet into several smaller botnets, or cells. As a result, a large botnet may be composed of thousands of such small cells. This makes it harder for an attacker to obtain a list of all active nodes. However, each bot has some (routing) information about other botnet nodes. Thus, an attacker who controls a large number of bots can identify a significant portion of the botnet. Also, as mentioned in the paper, the approach is vulnerable to host-based anti-virus software. If the anti-virus software detects that the host is part of a botnet, it could report the IP addresses of itself and other known hosts within the botnet to a central defense location. This may allow an attacker to obtain information about a large set of the small networks.

In [14], the authors describe a hybrid peer-to-peer botnet that partitions the botnet into servent bots and client bots. Servent bots use routing tables to store information about some other servent nodes in the botnet that they forward commands to. The robustness of the approach is built upon the idea that each node only knows about a few neighbors. Communication between servent hosts is performed using a custom-built, encrypted protocol. Similar to the army of botnets, each captured node reveals some information about other nodes in the network. Also, the connections between nodes use a non-standard protocol, and thus, could be detected at the network level.

4. OVERBOT

Overbot is a botnet communication protocol based on Kademlia. It provides a stealth means to send commands from the botmaster to compromised nodes. The design of Overbot is such that it is robust to the attacks outlined previously. Capturing nodes does not reveal any information about other nodes of the botnet. Also, the botnet leverages existing P2P networks to hide its communication among legitimate traffic. Finally, compromised nodes behave similarly to legitimate clients, and the botmaster never contacts a bot directly.

Similar to Peacomm, the Overbot protocol uses search requests in an existing P2P network (Kademlia) to have bots request commands from the botmaster. The difference to Peacomm is the way in which these search requests are crafted. In Overbot, each bot sends queries that only the botmaster can identify as command requests. At the network level, these requests appear indistinguishable from regular search queries. This is true even when the attacker controls several bots and is able to observe and record the requests that these captured nodes send. Obviously, this implies that every bot has to send different queries.

To craft search queries that request commands, Overbot makes use of a public-key model. That is, the botmaster owns a public-private key pair, and the public key of the botmaster is known to all compromised nodes. In order to communicate with the botmaster, botnet nodes encrypt messages with the public key of the botmaster. Only the botmaster, who possesses the private key, can identify these messages.

The following section discusses the way in which a compromised node requests a command. Then, we present the way in which the botmaster can respond and deposit a command.

4.1 How to request commands

To request commands, or, more general, to send messages to the botmaster, the nodes issue search requests in regular intervals. The target of each search request is a cipher text that is generated by *encrypting a sequence number with the public key of the botmaster*. For an outsider, these requests look like requests for legitimate hash values. However, the botmaster (who possesses the secret key) is able to recognize these requests. Moreover, once a message is decrypted, he can extract the node's sequence number. This sequence number is then used to send a command to the node.

Encoding information inside messages.

A node has to transmit two pieces of information in the search requests that it sends out. First, it has to secretly announce to the botmaster that the request is sent from a compromised node that is part of the botnet. Second, it has to request commands.

There are several options how information can be encoded within search requests:

The first option is the usage of the `FIND_NODE` and `FIND_VALUE` functions, which allow a node to search for a specific value inside the DHT. The hash value, which has a length of 160-bit, can be used by the node to embed information for the botmaster.

Second, it is possible to use the `STORE` RPC function in order to encode information within requests. However, some Kademlia implementations restrict the data that is accepted

by the `STORE` function. In eMule, it is possible to store a hash value and alternatively meta-data of a specific file. In Bittorrent, instead, only the own IP address and port number can be stored.

The third option to encode information within search requests is the node ID of the sender. A node ID has 160 bits, and each node can select its own ID is a way that some information can be transmitted to another node that receives a message.

Overbot uses the first option to request commands. That is, Overbot uses the 160-bit hash values that are part of search queries to transmit a sequence number that can be used by the botmaster to publish commands. Moreover, the protocol uses the node's ID to announce its botnet membership.

Node IDs and sequence numbers.

When a node is initialized, it generates its own node ID. In Overbot, this ID is generated by encrypting a hard-coded string with the public key of the botmaster. This string serves as a magic value, and the same string is used by all nodes within a botnet. Overbot uses the non-deterministic elliptic curve cryptography for encrypting this string. As a result, the same clear text is mapped to different cipher texts, ensuring that the node IDs of every bot will be different. Also, the space of possible node IDs is sufficiently large so that an attacker cannot enumerate all possible encryptions of the hard-coded string (which would allow a straightforward way to identify bots). If a deterministic algorithm such as RSA is used, a random padding would be required to guarantee different node IDs. The magic value allows the botmaster to identify a node as part of the botnet when he receives a message containing such a node ID.

Individual botnet nodes use sequence numbers, which allow the botmaster to predict the hashes of future search requests and to derive a symmetric key that can be used for messages that are sent to a node (as discussed in more detail later). At initial deployment, the sequence number is initialized to a random value. This ensures that different bots send different search requests. Afterwards, the number is increased by one after each time interval (e.g., after one minute). If the sequence number reaches the maximum value, it wraps back to zero.

Sending information to the botmaster.

To announce itself to the botmaster, a compromised node issues searches for DHT keys in regular intervals. These keys are generated by encrypting the current sequence number with the public key of the botmaster. Receiving such a message allows the botmaster to learn about the existence of the node as well as its current sequence number.

To be able to receive messages sent by the botnet nodes, the botmaster injects multiple *sensor nodes* into the DHT. These sensor nodes participate as regular clients in the P2P network. They are under direct control of the botmaster and store sensitive data such as the private key of the botmaster. The nodes are different from bots since they are not random, compromised machines but hosts owned by the botmaster. Sensor nodes do not produce any botnet-related traffic, and, thus, cannot be identified as malicious by an attacker.

Whenever a sensor nodes receives an incoming search request, it attempts to decrypt the node ID with its private key. If this decryption operation yields the magic value, it is very likely that the peer is part of the botnet. At this point, the sensor node adds the IP of the sending host to the list of compromised machines. Also, the sensor node extracts the current sequence number from the key (hash value) that the bot is searching for.

It is crucial to observe that a compromised node does not have any knowledge about the locations of the sensor nodes. Thus, it regularly issues requests for different search keys (determined by encrypting the increasing sequence numbers), with the "hope" that one of these requests will eventually reach a sensor node. Clearly, not every request will reach a sensor node. The botmaster has to ensure that he joins with sufficient sensor nodes such that the probability that a request will eventually reach a sensor node within a reasonable amount of time is high. This issue is discussed in more detail in Section 6.

4.2 Sending commands to botnet nodes

There are two different options how the botmaster may transmit messages to botnet nodes. The optimal choice depends upon the properties of the used DHT implementation. In eMule, it is possible to store nearly arbitrary values in the DHT. Therefore, the botmaster may store a message in the DHT, which can then be read by a botnet node. As Bittorrent restricts the values which can be written to the DHT, this technique is not possible in that case. However, the usage of an UDP-based DHT protocol makes it possible to use out-of-band methods, sending commands by using forged IPv4 source addresses.

Messages are sent to the botnet using actuator nodes. These nodes obtain their commands and the network location of the intended recipients by the botmaster. Unlike sensor nodes, actuator nodes actively issue requests to the network. Therefore, the risk of detection is higher than in the case of sensor nodes.

In-band messages.

When the botmaster receives a search request from a botnet node, he learns this node's current sequence number. As sequence numbers are increased at a particular, known rate, the botmaster is able to calculate the sequence number of the node at any later point in time.

If a deterministic public key algorithm would be used, the botmaster could directly predict the target ID of future search requests by encrypting the respective sequence number with its own public key. With non-deterministic algorithms such as elliptic curve cryptography (ECC), this is not possible. The reason is that the same sequence number can be mapped to different values. Therefore, the botmaster cannot predict the cipher text. To work around this problem, the botmaster and the node can both calculate the SHA-1 hash of the sequence number. This hash, which is deterministic, is then used as a *meeting point value*. More precisely, for each new sequence number, the botnet node issues an additional search request, using the meeting point value as key. If the botmaster wants to transmit a message to the node, he can encrypt the message (using the SHA-1 hash as a key) and then instruct an actuator node to store the message under the calculated hash value. Typically, this would store the command information on a legitimate client in the DHT. The bots would then contact this client via the search for the meeting point value and download the command. Of course, a checksum can be added to the command

to handle situations where, by accident, an unrelated data item is already stored under the meeting point value.

As messages are stored in the DHT for some time, it is not necessary to know the exact sequence number of a botnet node. The botmaster just needs to know a sequence number that will be looked up by the botnet node at some future point in time. Furthermore, from a technical point of view the interval in which sequence numbers are incremented may be arbitrarily large. As the output of ECC is non-deterministic, a botnet node does not need to change its sequence number in order to announce itself to the sensor nodes.

Out-of-band messages.

When Bittorrent is used, it is not possible to store commands for nodes directly in the DHT. The reason is that a node may only store IP addresses in the DHT. Furthermore, a token by the remote DHT node needs to be received first before an IP address can be stored. This prevents a host from using fake IPv4 addresses to store arbitrary values.

To send messages to a bot with Bittorrent, another mechanism needs to be used. If the botmaster controls a host that can use fake IP addresses and no egress filtering is in place, he can directly send information to the compromised machine. To this end, the message can be embedded within a normal DHT request (to prevent intermediary nodes or firewall software from identifying any suspicious requests). By using the same symmetric encryption mechanism as described for the in-band message case, and by storing the information inside the source node ID, this allows the botmaster host to use the full 160-bit available in the node ID when transmitting information. Because a spoofed source IP address is used, the botmaster cannot be traced back easily.

5. ENCRYPTION

Overbot communication is encrypted by means of public key and symmetrical cryptography. When requesting commands, parts of the messages (i.e., the key of search requests) sent by regular nodes are encrypted with the public key of the botmaster. Commands that are deposited (or directly sent) by the botmaster are encrypted with a symmetric key, which is derived from the sequence number. This ensures that an attacker is not able to identify botnet nodes by looking at the content of these messages. In this section, we first discuss the reasons why elliptic curve cryptography (ECC) has been chosen for public key cryptography. Afterwards, we outline the algorithm that is used to encrypt messages from the botmaster to the compromised nodes.

5.1 RSA vs. ECC

A requirement on the used encryption algorithm is that it should be possible to store encrypted data in the various 160-bit fields that are used by Kademlia. This requires that the length of the cipher text generated by the encryption algorithm does not exceed 160-bit.

The obvious first choice for a public key algorithm is RSA. When RSA is used, a cipher text with a length of 160-bit limits the value of n in the public key (n, e) to a maximum size of 160-bit. Unfortunately, a 160-bit number can be factored in less than an hour on a modern computer. By factoring n, the attacker is able to calculate the private key of the botmaster. Therefore, RSA cannot be used for the required purpose.

An alternative to the RSA algorithm is elliptic curve cryptography (ECC). One property of elliptic curve cryptography is that it offers security comparable to RSA, at much smaller key sizes. According to the NSA, an 160-bit ECC key is roughly comparable to a 1024-bit RSA key [9]. As of March 2008, the largest ECC key which has been cracked had a size of 109-bit. It was cracked using 10.000 Pentium class PCs, working for 540 days [1].

The SECG secp112r1 [2] curve has a size of 112-bits and provides about 56-bits of security. If a plain text of up to 40-bits is used, the size of the cipher text has a maximum value of 160-bits. While it is possible that keys based on the secp112r1 curve can be cracked in the future, the security provided by the secp112r1 curve is sufficient for our purposes. It should be noted, however, that the Standards for Efficient Cryptography Group (SECG) compares the security of secp112r1 to 512-bit RSA. A 512-bit RSA key had already been factored in 1999 [10].

5.2 Encrypting messages from sensor nodes

When the botmaster has knowledge about a node's current sequence number, he is able to use this information to derive a symmetric key. This key can be used to protect messages intended for the botnet nodes. An option would be to use the SHA-1 hash of the sequence number as a symmetric key. The length of the key is equal to the length of the plain text, and each key is only used once. Therefore, the cipher text could be obtained by using the XOR combination of the plain text with the key. Encrypted messages need to include a checksum to allow botnet nodes to verify that the message has been encrypted with a given key.

When the sensor node intends to decrypt a message, it tries to decrypt the message with the set of key values that are likely to have been used to encrypt a message. The size of the set of possible keys depends on the time intervals used to increment sequence numbers. The botnet node can verify successful encryption by validating an included checksum.

If the range of valid sequence numbers is small and other information, such as the node ID, is available to the botmaster, it is beneficial to also include this information in the generation of the SHA-1 hash. This prevents an attacker from being able to pre-compute all possible SHA-1 values and then check the hash values of incoming requests against these pre-computed values.

Care must be taken that the SHA-1 input range for the generation of the symmetric key and the SHA-1 input range for the calculation of the meeting point do not overlap, e.g., by negating the sequence number in one of the cases. Otherwise, it is trivial for an attacker to decrypt messages sent by the botmaster.

6. PROBABILITY OF RECEIVING REQUESTS

An important consideration for the feasibility of Overbot is the time that it takes for a sensor node to learn about the existence of a compromised machine. In this section, we estimate the probability that at least one sensor node (and, as a result, the botmaster) learns about the existence of a compromised node within a certain time interval. To this end, we propose a model that can be used to calculate the

probability of at least one query hitting a sensor node within this interval. The model requires the knowledge of certain parameters of the used P2P application. These parameters include the numbers and the types of queries that are sent within a given interval, as well as the total number of nodes present in the network.

Using our model, we can estimate the probability that a compromised node can announce its presence and retrieve commands. To validate this model, we have implemented the Overbot protocol and performed experiments on the actual Bittorrent DHT network. Moreover, we discuss several optimizations that allow a bot to contact sensor nodes faster. If it is not required that a node conforms perfectly to a legitimate DHT node, it is possible to reach the desired goal much faster than with a faithful DHT implementation. The drawback is that such atypical behavior could be used by an attacker to distinguish between legitimate and compromised nodes.

6.1 Botnet model

The Overbot model is used to estimate the probability that at least one request of a particular botnet node is received by a sensor node. We model a botnet node as an entity that sends packets at a given rate to other nodes within the DHT. The probability that a request sent by a botnet node is received by a sensor node can be modeled as a Bernoulli trial. Therefore, the probability that, after n requests, k sensor nodes have been contacted, can be modeled with the binomial distribution:

$P(k) = \binom{n}{k} p^k (1-p)^{n-k}$

where n is the total number of requests by a particular node and k is the number of successes with a probability of p. For the probability that after n requests at least k requests have been received by the sensor nodes, the above formula can be used:

$P(X \geq k) = \sum_{x=k}^{M} \binom{n}{x} p^x (1-p)^{n-x}$

where M denotes the number of sensor nodes. To calculate the probability that, after n requests, at least one sensor node has been contacted, the simplified formula $1-(1-p)^n$ can be used.

The expected value $E(X)$ of the binomial distribution is given as $E(X) = n \cdot p$. That is, after n Bernoulli trials, each with a probability of p, we expect $n \cdot p$ successes.

As a simplification, we assume that the probability of a single packet being received by a sensor node is $\frac{M}{N}$, where N is the total number of nodes and M is the number of sensor nodes. Especially for nodes that have only been a member of the DHT for a short time, this assumption will be invalid. In this case, only few other peers will know about the new node. Therefore, the node will receive less incoming queries than other peers. Furthermore, nodes in the local routing table have a higher chance of receiving requests than other random nodes.

6.2 Measurements

For our model, we need to determine the number of nodes N in the Bittorrent DHT (to determine $p = \frac{M}{N}$), as well as the number of requests n that are typically issued to locate a particular key. For these measurements, we built a modified Bittorrent DHT client. This client iteratively issues search requests for random infohashes. Directly after the DHT implementation marks a query as finished, a new query is started.

The first test was used to estimate the current size of the Bittorrent DHT. The Khashmir library is able to estimate the size by calculating $K * 2^{n_{buckets}-1}$. In the Khashmir Kademlia implementation, K has the value 8. After an uptime of approximately ten hours, the client reported an estimated DHT size of $8,388,608$.

In the following tests, the number and types of outgoing packets sent by a peer were recorded. For this experiment, the same, modified Bittorrent client was used. After an uptime of two hours, a tcpdump process was started, capturing all network packets for about 140 minutes. The delay of about two hours is required because packets sent during the bootstrap phase would otherwise modify the results.

According to the network traffic capture file, 119 different infohashes were looked up by the local node. On average, a lookup required 45 `get_peers` requests to other nodes of the DHT. A total of 69,328 packets were recorded. 36,806 packets (53%) were outgoing packets, and 32,522 (47%) packets were incoming packets. Of the 36,806 outgoing packets, 10,961 (30%) were queries and 25,768 (70%) were responses to remote queries. Most of the remaining packets included error messages to remote peers that sent invalid tokens. The 10,961 queries were sent to a total of 7,433 unique IP addresses. The queries can be further subdivided into 4,575 (42%) `find_node` requests, 1,034 (9%) ping requests, and 5,351 (49%) `get_peers` requests. The 5,351 `get_peers` requests were sent to 4,355 unique IP addresses. Therefore, about 20% of the `get_peers` requests were sent to nodes in the local routing table.

6.3 Experiment

Using our model, together with the parameters that we determined for the Bittorrent DHT, we can estimate the expected number of queries that a compromised machine has to send before a sensor nodes receives a message. To validate this model, we built a client application based on the Bittorrent DHT library that iteratively issued search requests for random IDs to the DHT.

The tests were performed using a total number of 10 sensor nodes (that is, $M = 10$) and 10 bot nodes (that were querying for random keys). The total time of the test was 4,150 minutes (slightly less than 3 days). During this time, 34,918 different infohashes were requested. When multiplying this number by 45 (for the number of `get_peers` requests for each infohash - see the previous section), and then subtracting the 20% of queries that are sent to nodes in the local routing table, this results in about 1,250,000 `get_peers` requests that have been sent (roughly 125,000 by each bot node). During our experiment, a total of five `get_peers` requests were received by the sensor nodes.

By applying our model to the numbers mentioned above, we expect the total number of received `get_peers` requests to be $E(X) = n \cdot p$, where n is the total number of requests and p is the probability that a request is received by a sensor node. For our test setup, with approximately 1,250,000 requests and 10 sensor nodes, the expected value is $E(X) = 1,250,000 \cdot \frac{10}{8,388,608} = 1.49$. This is somewhat lower than the five requests that were actually received.

There could be several reasons why the number of experimentally received requests is higher than the predicted number. As the size of the DHT is an approximation, this introduces inaccuracies into the model. Another factor is that the uptime of our nodes may be higher than the uptime of

other DHT peers. Therefore, our nodes may receive a higher percentage of incoming requests than other nodes. However, as the experimental results are within the same magnitude as the predicted results, this shows that the proposed model is applicable to the problem.

If a 90% probability is desired that at least one packet of a particular botnet node is received by one of the M sensor nodes, this can be calculated by deriving the formula of the binomial distribution to $n = \frac{log(1-0.9)}{log(1-M/8{,}388{,}608)}$. For 100 sensor nodes, this would result in about 193,000 requests. These requests would require about 4.4 days on the setup used for the measurements. For 1,000 nodes, this time would drop to less than 12 hours. Thus, it is entirely feasible to locate most compromised nodes within a day, using a modest amount of sensor nodes that are injected into the DHT.

6.4 Optimizations

The model above assumes that the compromised nodes behave exactly as other client nodes. However, several optimization techniques are possible to increase the probability that a query is received by a sensor node.

The first, simple optimization is the removal of the rate limiter, which limits the maximum outgoing bandwidth to one kilobyte per second. In this case, many more requests can be sent out every second. A original Bittorrent node would transmit about ten outgoing UDP packets per second. When assuming that the modified DHT implementation is issuing ten `find_node` requests per second, the chances of hitting a sensor node would be significantly higher. While the original implementation, which issued 31 `get_peers` requests per minute, required about 4.4 days for a 90% probability of hitting a sensor node, an implementation which is able to issue 600 `get_peers` requests per second would only require 5.3 hours.

Furthermore, `find_node` requests could be utilized by a node in order to announce itself. The DHT specification states that when a bucket has not changed for a least 15 minutes, the node should issue a `find_node` request for a random ID in the range of the bucket. This ID could be generated by encrypting the sequence number with ECC. This encryption needs to be repeated until the resulting cipher text is located within the required range.

The disadvantage of the aforementioned optimizations is that a node does not exhibit the same behavior as a real Bittorrent client anymore. Therefore, an observer may be able to discover nodes that are members of the botnet.

7. ATTACKS / COUNTERMEASURES

In this section, we discuss possible countermeasures and attacks against the Overbot protocol. In particular, we observe that bot nodes and sensor nodes could be identified based on behavioral differences that distinguish them from legitimate nodes when observed for a longer period of time. An attacker who is able to identify different behaviors between a botnet node and a legitimate client might be able to detect that a given host on the network is part of the botnet. This allows the attacker to create a blacklist of infected hosts.

One problem for Overbot stems from the fact that an attacker can launch statistical attacks that look at the behavior of nodes. For example, a single botnet node communicates with many other host in the DHT network. Thus, when one single nodes is issuing a lot of requests, the node might be part of the botnet. To perform this kind of detection, more research is required to obtain better models that can describe the expected behavior of normal nodes in P2P networks, as significant or persistent deviations could provide indications of infected hosts.

Another problem stems from the fact that the botmaster needs to control a large number of node IDs. If a single IP address or net block is used for all the connections by the botmaster, an attacker might be able to identify these IPs.

Finally, when analyzing the DHT provided by Bittorrent, we observe an important design decision that should be considered when implementing future network protocols: only allow to store information that is actually required, and check as much information as possible. In Bittorrent, the value of a DHT entry may only include the IP addresses and port numbers of peers that are part of the torrent swarm. If an IP address is stored, the host that wants to store the value must be able to receive DHT packages at this IP address. This ensures that it is more difficult to inject fake information in the DHT. Thus, when designing new protocols, care should be taken to restrict the amount of freedom that nodes have when choosing identifiers or embedding data in messages.

There is a tradeoff between the length of the intervals after which the sequence number is incremented and the security of the botnet nodes. If the botmaster decides to store information for the nodes in the DHT, a long time interval might allow an attacker to better predict the hashes used to store information. This allows the attacker to join the DHT with a node ID close to the predicted hashes. By observing the `STORE` requests to his own node, he can, therefore, obtain the network address of the botmaster.

8. CONCLUSIONS

This paper presents the design of Overbot, a botnet protocol based on Kademlia. One of the main goals was to improve the robustness and stealth of the command and control channel compared to existing botnet protocols. In particular, even an attacker with control over the network and many bot nodes is not able to definitely assert if a node is part of the botnet. The usage of an existing P2P protocol makes traffic analysis harder, as the direct neighbors of a node are likely to be legitimate DHT peers. Queries issued by Overbot look similar to legitimate search requests, which makes filtering harder.

P2P botnets can be much more resilient to attacks than traditional client-server networks. If existing P2P protocols are used, it is hard to distinguish a legitimate node from a malicious one. The purpose of developing Overbot is to anticipate future developments of advanced command and control channels. The hope is that the research community can find solutions before such botnets appear in the wild.

A promising approach to attack botnets such as Overbot is to create models that capture the behavior of legitimate nodes. If a node acts differently, over a longer period of time, this might be an indication of a malware infection. For example, issuing a lot of requests for non-existing hashes might be a sign that this node is part of the botnet. Also, a major issue in the design of new P2P protocols is to prevent botnets from utilizing their infrastructure. Thus, more research in the area of P2P botnets needs to be carried out, as we are likely to see more of these botnets in the real world.

9. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful comments on improving this paper.

This work has been supported by the Pathfinder project in FIT-IT Trust in IT Systems, the Austrian Science Foundation (FWF) and by Secure Business Austria (SBA) under grants P-18764, P-18157.

10. REFERENCES

[1] Certicom. Press Release: Certicom Announces Elliptic Curve Cryptosystem (ECC) Challenge Winner. `http://www.certicom.com/2002-press-releases/38-2002-press-releases/340`, Nov. 2002.

[2] Certicom Research. SEC 2: Recommended Elliptic Curve Domain Parameters. `http://www.secg.org/download/aid-386/sec2_final.pdf`, Sept. 2000.

[3] D. Dagon, G. Gu, C. Lee, and W. Lee. A Taxonomy of Botnet Structures. In *Annual Computer Security Applications Conference (ACSAC)*, 2007.

[4] J. Douceur. The Sybil Attack. In *International Workshop on Peer-to-Peer Systems*, 2002.

[5] J. Grizzard, V. Sharma, C. Nunnery, B. Kang, and D. Dagon. Peer-to-peer botnets: Overview and case study. In *First Workshop on Hot Topics in Understanding Botnets*, 2007.

[6] A. Loewenstern. Bittorrent DHT Protocol. `http://www.bittorrent.org/beps/bep_0005.html`, Jan. 2008.

[7] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *International Workshop on Peer-to-Peer Systems*, 2002.

[8] E. Michelangeli. KadC (P2P library) Documentation. `http://kadc.sourceforge.net/`, Oct. 2006.

[9] National Security Agency (NSA). The Case for Elliptic Curve Cryptography. `http://www.nsa.gov/ia/industry/crypto_elliptic_curve.cfm`, Mar. 2008.

[10] RSA Laboratories. Announcement: RSA-155 is factored! `http://www.rsa.com/rsalabs/node.asp?id=2098`, Aug. 1999.

[11] R. Schoof and R. Koning. Detecting peer-to-peer botnets. `http://staff.science.uva.nl/~delaat/sne-2006-2007/p17/report.pdf`, Feb. 2007.

[12] J. Stewart. Phatbot Trojan Analysis. `http://www.secureworks.com/research/threats/phatbot`, Mar. 2004.

[13] R. Vogt, J. Aycock, and J. M. J. Jacobson. Army of Botnets. In *Network and Distributed System Security Symposium (NDSS)*, 2007.

[14] P. Wang, S. Sparks, and C. Zou. An advanced hybrid peer-to-peer botnet. In *First Workshop on Hot Topics in Understanding Botnets*, 2007.