Master's Thesis

A Comparative Analysis of the Resilience of Peer–to–Peer Botnets

Dennis Andriesse
Amsterdam, August 2012

VU University Amsterdam

Advisors: Herbert Bos and Christian Rossow

Abstract

Botnets have traditionally used centralized architectures for command and control. In such architectures, a relatively small number of centralized servers is used to command the bots. Centralized botnet architectures are straightforward to deploy, but relatively easy to take down by disabling the command and control servers. In an effort to increase the resilience of their botnets, malware creators have begun to implement peer–to–peer command and control architectures. In peer–to–peer botnets, rather than relying on centralized command servers, bots cooperate to spread commands amongst themselves. This lack of centralized command and control servers potentially makes peer–to–peer botnets very difficult to disable, but the exact degree of resilience greatly varies depending on the communication model and architecture of each botnet. In this thesis, we analyze and compare the resilience of the major peer–to–peer botnet threats to date. Additionally, we present the results of our reverse engineering analysis and takedown attempt against Zeus, a novel and previously undocumented peer–to–peer botnet.

Contents

Chapter 1

Introduction

1.1 Context and motivation

Traditional botnets are controlled via centralized architectures. We refer to the individuals controlling a botnet as *botmasters*. Most commonly, traditional centralized bots receive their commands via IRC chat channels, or download commands from HTTP servers. Any channel used to command a botnet is referred to as a *command and control* (C&C) channel.

Although centralized C&C architectures are quite easy for botmasters to deploy, they are also relatively easy to disable by taking down the centralized C&C servers, often through legal action or blacklisting approaches. If a botnet's C&C servers are disabled, the botnet is rendered useless to the botmasters. Multiple centralized botnets have been successfully disabled through the takedown of their centralized C&C servers [1].

To overcome the vulnerability of centralized C&C servers, botnets using peer–to–peer (p2p) architectures are becoming increasingly popular among malware creators. Botnets using such architectures can be very difficult to attack, due to their decentralized nature. How resilient a p2p botnet is depends on the communication model and architecture it uses, but well designed p2p networks can be extremely resilient even when confronted with many disabled or poisoned bots [2].

Several p2p botnets have appeared in the wild already. Some of these have been successfully taken down, but others remain quite resilient. Most notably, the Sality p2p botnet has operated largely undisturbed since 2008 [3]. A number of theoretical studies have shown that future p2p botnets could become even more resilient than those seen in the wild so far [4, 5].

In this thesis, we analyze and compare the resilience of the major peer–to–peer botnet threats to date. To supplement our comparison, we present the results of our reverse engineering analysis and takedown attempt against Zeus, a novel and previously undocumented peer–to–peer botnet.

1.2 Structure of this thesis

In Chapter 2, we provide an overview of the major p2p botnet threats to date. For each of the botnets discussed, we also review any documented vulnerabilities.

As an in–depth study of a modern p2p botnet, Chapter 3 presents the results of our reverse engineering analysis of Zeus. The Zeus p2p botnet first appeared around October 2011, and our reverse engineering study is the first to evaluate it in detail.

We also describe the results of a first takedown attempt against the Zeus p2p botnet, which we executed in April and May of 2012. These results are described in Chapter 4.

Chapter 5 provides a comparative analysis of the resilience of the major p2p botnets to date. We identify which botnets are most successful in the wild, and derive several characteristics that contribute to their success. Furthermore, we estimate the resilience of each of the compared botnets to peerlist poisoning, one of the most promising and generalized attack vectors against fully decentralized p2p botnets.

Chapter 6 discusses related work. Finally, we present our conclusions in Chapter 7.

Chapter 2

An Overview of Peer–to–Peer Botnets

In this chapter, we provide an overview of the most significant p2p botnet threats to date. The botnets are described in order of the dates at which they were first seen in the wild. For each botnet, we briefly describe the most relevant aspects of its architecture and communication model. Additionally, we review each botnet's documented vulnerabilities, if any.

2.1 Storm

Storm is a p2p bot which first appeared in January 2007 [6]. The Storm botnet was mainly used for spamming, and was active until December 2008, after which it fell into disuse [7]. At its peak, the Storm botnet is estimated to have contained around 80.000 bots [8].

Communication protocol

Storm is based on Overnet, a Kademlia implementation [9]. Overnet was originally designed for the eDonkey file sharing application. In 2006, following legal issues, the eDonkey creators agreed to cease their operations, and the eDonkey application was discontinued. However, the Overnet network remained active and in use by alternative file sharing clients. The Storm botnet later used this remaining Overnet network to bootstrap its operations.

In what follows, we provide a brief overview of how Storm bots communicate using the Overnet protocol. The information provided here is based on the works of Porras et al. [6] and Holz et al. [8].

Storm communicates using the binary Overnet protocol over UDP. Each bot uses a random high–order UDP port for communication. Storm Overnet packets can be distinguished from other Overnet packets by their network identifier `0xE3`, which is present in the header of each transmitted packet.

Each Storm bot has a randomly generated 128 bit identifier, created when Storm is first run. This identifier is used to compute how "close" a particular bot is to a given key. The distance between two identifiers or keys is determined using the Kademlia XOR–metric. The Kademlia XOR distance d between two keys x and y is computed as $d = x \oplus y$, where d is interpreted as an integer.

Storm bots contain hardcoded lists of 200 to 900 bootstrap peers. New Storm bots make themselves known in the network by sending Overnet `PUBLICIZE` packets to their bootstrap peers, causing the bootstrap peers to consider them for addition to their peerlists.

Bots organize their peerlists into 128 "buckets", where bucket i, with $0 \leq i < 128$, contains peers with distance $2^i \leq d < 2^{i+1}$. Each bucket can contain up to 20 peers. Since the numeric distance between 2^i and 2^{i+1} grows with increasing i, bots store many peers close to themselves, while they store relatively few peers far from themselves. The peers in each bucket are kept sorted by the time of last contact.

Storm bots learn about new peers through incoming `PUBLICIZE` packets, and by sending Overnet `CONNECT REQUEST` packets to other peers. In response to a `CONNECT REQUEST`, a peer returns a list of 20 other peers close to its own identifier.

Bots which learn about new peers update their peerlists as follows. If the identifier of the

new peer is already known, then the IP address and port for that peer are updated and the peer is given a fresh timestamp. Else, if the bucket that the new peer belongs in is not yet full, then the new peer is added to the peerlist. If the bucket is full, then the oldest peer from the bucket is sent a `PING` message to test it for responsiveness. If it is still responsive, then the new peer is discarded and the old peer's timestamp is updated. If the old peer is no longer responsive, then it is removed and the new peer is added.

Storm bots locate commands by searching for special keys in the network. The keys under which commands can be located are computed using a time–based algorithm. Storm bots search for a key in the network by sending `ROUTE REQUEST` packets containing the search key to the peers they know that are closest to the key. These peers then return `ROUTE REPLY` messages containing the peers they know that are close to the key being searched for. These new peers are in turn contacted with `ROUTE REQUEST` messages to find peers increasingly close to the search key. This process continues iteratively until a `ROUTE REPLY` is received which contains peers further away from the key than the peer returning the reply, in which case the key is found.

Storm originally shared the Overnet network with benign file sharing clients. In October 2007, the Storm authors modified Storm to use an XOR encryption algorithm on its messages, effectively separating the Storm bots from the original Overnet network into an encrypted Overnet network consisting exclusively of Storm bots.

Vulnerabilities

Storm bots retrieve commands by looking up *command keys*. These are Overnet identifiers periodically generated by each Storm bot using a time–based algorithm. The Storm command key algorithm generates 32 command keys per day, which each Storm bot can use to find the latest commands [6, 8]. Since the Storm botmasters also know the command key algorithm, they can publish commands under the correct keys during each time period.

The main weakness of Storm lies in the fact that anyone who successfully reverses the command key generation algorithm can determine in advance where the commands of the next day will be published. Because the published commands are not authenticated by the Storm bots, attackers can overwrite any commands as soon as they are published, rendering the Storm botmasters unable to command their botnet [8].

2.2 Sality

The Sality malware family has been around for a long time. It started out with a simple data stealing virus in 2003. Since then, Sality has developed into a sophisticated p2p botnet, which first appeared in January 2008. The latest versions of Sality are mainly intended to drop additional malware [3].

The Sality p2p botnet has seen several major updates. Currently, two Sality versions are still active. The largest network is formed by Sality v3, which stems from 2009. Its size is currently estimated at around 200.000 bots. Sality v4 is very similar to the v3 variant, but fixes a critical vulnerability, which we will discuss later. The v4 Sality network is growing, and is expected to eventually become the dominant Sality network [3].

As Sality v3 and Sality v4 are very similar, we discuss them both in the rest of this section.

Communication protocol

The information described here is based on the Symantec technical report on Sality [3]. All details discussed apply to both Sality v3 and Sality v4, unless noted otherwise.

The Sality p2p network uses a simple but highly robust custom binary protocol. Messages are exchanged over UDP, and are encrypted with RC4, using the first 4 bytes of the message payload as the key. Bots attach themselves to a pseudorandomly generated port, based on the computer name. The Sality network is unstructured, and bots exchange commands in a gossip–like fashion. Commands are exchanged in the form of signed URL packs, which contain URLs where the bots are to download additional malware. In addition to the exchange of these URL packs, Sality v4 is able to exchange signed binaries via the p2p

network, eliminating the need for centralized hosting of the binaries.

Sality bots establish initial contact with the network through a hardcoded list of bootstrap peers, which are copied to a local peerlist when Sality is first executed. The local peerlist has a maximum size of 1000 peers.

Sality peerlists contain an identifier, IP address and port for each known peer, as well as the time the peer was last contacted, and a per–peer *goodcount* value. The goodcount value is especially interesting. It is an integer value which indicates how trustworthy a peer has proven to be in the past, so that bad or fake bots can be recognized and removed from the peerlist. Goodcount values are maintained locally by each peer, but are never exchanged over the p2p network.

Sality identifiers are plain integers. Sality bots start out with their identifiers set to zero, and are assigned identifiers later by other bots, depending on whether or not they are externally reachable.

Sality bots contact all their known peers every 40 minutes. A Sality bot initiates communication with another bot by issueing a `pack exchange` query, which contains the sequence number of the current URL pack that the requesting bot has. If the remote peer does not respond to the query, or replies with a bad response, then the requesting peer decrements the goodcount value of the remote peer and terminates communication with it. If the remote peer returned a good reply, its goodcount is incremented. Peers with goodcounts below -30 are removed from the peerlist if the peerlist length is at least 500.

If the remote peer is responsive and has a newer URL pack than the requesting peer, it returns this URL pack to the requesting peer. Otherwise, it returns an `ack` message, indicating whether its URL pack sequence number is equal to or lower than that of the requesting peer. If the requester sees that the remote peer has an older URL pack sequence number, then it sends its own URL pack to the remote peer.

Next, if the requesting peer notices that its identifier is still set to zero, it requests an identifier from the remote peer. The remote peer then attempts to contact the requesting peer using a `pack exchange` query. If this succeeds, the remote peer concludes that the requesting peer is externally reachable, and returns an identifier $\geq 16.000.000$ to the requester. In this case, the requesting peer is also added to the remote peer's peerlist. Since this is the only way for new peers to make it into other peers' peerlists, Sality peerlists contain only externally reachable peers. If the requesting peer turns out not to be externally reachable, it is assigned an identifier $< 16.000.000$.

Finally, if the requesting peer has a peerlist shorter than 980 peers, it requests an additional peer from the remote peer. The remote peer returns a single peer with positive goodcount, chosen randomly from its peerlist.

Vulnerabilities

The Sality v3 network has a serious vulnerability. Namely, while the URL packs exchanged via the p2p network are signed, the binaries downloaded from the URLs provided in these URL packs are not signed. This means that if an attacker is able to take over one of the domains exchanged in the URL packs, it is possible to make the Sality bots download a binary which will disinfect them. This vulnerability has been fixed in Sality v4, which uses signed binaries as well as signed URL packs [3].

2.3 Waledac

Waledac is a p2p bot which first appeared in April 2008. Analysis suggests that it is the successor of the Storm bot. The timing of Waledac's arrival supports this idea, since the Storm botnet fell into disuse in late 2008 [10].

Waledac is meant mainly to send spam and drop additional malware [11]. At its prime, the Waledac botnet is estimated to have contained around 165.000 bots [10]. The Waledac botnet was disabled in February 2010.

Communication protocol

The information described in this section is based on the analyses by Stock et al. [10], Tenebro [11] and Sinclair et al. [12].

Waledac uses a custom XML–based protocol. Bots communicate using HTTP over TCP. Apart from the plaintext HTTP headers, Waledac messages are encrypted using 128 bit

AES. The AES key is determined through an RSA encrypted key exchange.

Each Waledac bot generates an RSA public and private key pair when it is first run, and uses the generated private key to create a self–signed X.509 certificate. When a Waledac bot wishes to establish encrypted communication with another peer, it sends its self–signed certificate to this peer. The remote peer then uses the public key from the certificate it received to send back an RSA encrypted AES key. This AES key is then used to encrypt further communication between the two peers.

Waledac peerlists contain an IP address, port, time of last contact, and 20 byte identifier for each peer. Only externally reachable bots are kept in the peerlists of other bots. These externally reachable bots are referred to as *repeaters*. Bots check if they are externally reachable during their initial bootstrapping phases, and if not, they do not attempt to propagate themselves into the peerlists of other bots. Each Waledac bot contains a hardcoded list of 50 initial peers, which are used for bootstrapping onto the p2p network.

To obtain new peers, Waledac bots periodically send `peerlist exchange` requests to the peers already in their peerlists. During a peerlist exchange, both bots involved in the exchange select 200 random peers from their peerlists, and send these to the other bot. Both bots then incorporate the new peers into their peerlists, discarding older entries in case the maximum peerlist size of 1000 is exceeded. As a backup measure, Waledac bots are capable of downloading signed peerlists via a hardcoded URL hosted in a fast–flux network run by the Waledac bots themselves.

Waledac is not purely p2p–based. Instead, it relies on a relatively small number of backend servers which are used to spread commands to the bots. A bot trying to retrieve new commands contacts a repeater peer, which in turn relays the command request to one of the backend servers. The reply from the backend server is then routed back to the requesting bot via the repeater peer. Thus, above the p2p layer of Waledac, there is a centralized core, which the p2p layer serves to conceal.

Vulnerabilities

Although Waledac uses a p2p protocol to exchange lists of repeaters between the bots, it relies on a small number of backend servers to serve new commands to the network through the repeaters. Effectively, Waledac's p2p layer is just an obfuscation layer to conceal its centralized core.

In February 2010, Waledac was taken down by Microsoft[1]. The attack involved disabling the domains of Waledac's backend servers through legal action, while simultaneously poisoning the peerlists of the Waledac bots to prevent them from updating to new domains. A detailed description of this attack strategy against Waledac is available in [12]. The attack made it impossible for the botmasters to deliver new commands to Waledac, effectively rendering the botnet unusable.

2.4 Conficker C

Conficker C is the first Conficker variant to contain p2p functionality. To this day its exact purpose remains unknown. Conficker C first appeared in February 2009, and at its peak the Conficker C network contained around 200.000 bots. It was never really taken down, and is estimated to still contain around 25.000 bots[2].

Communication protocol

The information in this section is based on the technical report by Porras et al. [13].

Conficker C uses a custom binary protocol. It uses both UDP and TCP sockets. The message types exchanged over both types of socket are very similar to each other, and are encrypted using RC4. Conficker uses a Domain Generation Algorithm (DGA) as its backup channel, which it can use to download signed binary updates.

Conficker mainly uses its p2p network to upgrade itself to the latest binary version. When another Conficker bot is found, a connection is established to this newly found bot. The initiator of the connection advertises its current binary version to the remote peer. If the remote peer has an inferior binary version, it downloads

[1] `http://blogs.technet.com/b/microsoft_on_the_issues/archive/2010/02/24/cracking-down-on-botnets.aspx`
[2] `http://www.confickerworkinggroup.org/wiki/pmwiki.php/ANY/InfectionTracking#toc8`

the binary of the initiator of the connection. If the remote peer has a superior binary version, then it sends its binary to the initiator of the connection. If both peers have the same binary version, then the connection is terminated.

Both the binaries and the version numbers exchanged by Conficker bots are digitally signed using RSA, meaning that attackers can not insert rogue binaries into the Conficker network. The exchange of new binaries is the only means of C&C used by Conficker.

An interesting aspect about Conficker C bots is that they do not contain any hardcoded bootstrap peers. Instead, bots find new peers by scanning the Internet for other Conficker infections. In addition, Conficker has the ability to exchange peers by piggybacking a variable number of peers with its messages. Conficker peerlist entries contain an IP address, port, and an 8 byte identifier for each peer. Peers generate their identifiers randomly when they are first run.

A peer found via scanning is added to the scanning peer's peerlist only if it has the same binary version as the scanning peer. A maximum of 2048 peers are stored in Conficker's peerlist. If the peerlist is full and a new peer is discovered, then the oldest entry from the peerlist is overwritten with the new peer.

While scanning the Internet for other peers, Conficker probabilistically adds existing peers from its peerlist to its scanning targets. This is the only means for Conficker to recontact peers already in its peerlist.

2.5 ZeroAccess

ZeroAccess is a p2p botnet which has been around since June 2009. It is used for dropping malware, and is currently estimated to contain around 150.000 bots.

Communication protocol

The information in this section is based on the ZeroAccess technical report by Wyke [14].

ZeroAccess uses a custom binary protocol, which it runs over TCP. Messages are encrypted using RC4 with a hardcoded key. The sole purpose of the p2p network is to drop new binaries, and these binaries form the only means of C&C used by ZeroAccess.

ZeroAccess bots contain hardcoded peerlists of 256 bootstrap peers. ZeroAccess peerlists have a maximum length of 256, so that each bot starts out with a fully saturated peerlist.

Each peer in a ZeroAccess peerlist has an IP address, a port, and a time of last contact, but no identifier. ZeroAccess bots periodically traverse their peerlists, attempting to establish a TCP connection to each of the peers.

When ZeroAccess successfully connects to another peer, it first requests a peerlist from that peer. The remote peer then returns its *entire* peerlist of length 256. The requesting peer merges the received peerlist into its own peerlist by overwriting old peers with newer ones as much as possible. The fact that ZeroAccess bots accept so many new peers at once, and trust the timestamps provided by the responding peers, means that the peerlists of ZeroAccess bots can be poisoned quite effectively. To our knowledge, however, such an effort has not yet been made.

After downloading a new peerlist from a remote peer, ZeroAccess requests a list of binaries that the remote peer currently has available for downloading. If the requesting peer finds that the remote peer has one or more new binaries available, it downloads these binaries from the remote peer.

The binaries exchanged by ZeroAccess are signed using RSA, but it should be noted that a 512 bit key is used. This is considered rather short by today's standards. It is conceivable that this 512 bit key could be factored, so that rogue binaries could be injected into the ZeroAccess network.

2.6 Hlux

Hlux is the successor of the Waledac p2p botnet. It first appeared in December 2010, a few months after the Waledac botnet was taken down. Similar to the Waledac botnet, spreading spam is one of the main activities of Hlux. Additionally, Hlux is used for Denial of Service attacks and data theft.

Hlux was taken down through a poisoning effort by Kaspersky Labs in March 2012, but

[3] http://www.securelist.com/en/blog/655/Kelihos_Hlux_botnet_returns_with_new_techniques

has since restarted in the form of a fresh Hlux network[3]. The original Hlux network is estimated to have contained around 49.000 bots, while the respawned Hlux network currently contains around 130.000 bots.

Communication protocol

Our description of Hlux is based on the analysis of Werner [15].

Hlux uses a custom binary protocol over TCP. It encrypts its messages using three layers of encryption. Messages are first encrypted with Blowfish, then with Triple DES, and finally with another layer of Blowfish. Each of the three stages uses a different key.

Hlux uses an architecture similar to that of Waledac. Just as in Waledac, the network is ultimately controlled by a set of centralized servers, which we refer to as *control servers*. Externally reachable Hlux bots forward command requests from other bots to these control servers. This means that just as in Waledac, the Hlux p2p network is largely an obfuscation layer for the centralized core of the botnet.

Another important function of the Hlux p2p network is that it allows the botmasters to dynamically update the list of control servers in case any of the control servers are taken down. Like Waledac, Hlux also uses a set of hardcoded URLs hosted in its own fast–flux network as a backup channel.

Hlux bots come with around 200 initial bootstrap peers. Each bot has a randomly generated 16 byte identifier. Hlux bots store at most 500 peers in their peerlists, and keep a timestamp for each peer, indicating the time of last contact with that peer.

Whenever an Hlux bot contacts another bot, it actively pushes a list of its 250 most recent peers to the remote peer. In turn, the remote peer sends 250 of its peers back to the initiating peer. Both peers update their peerlists, keeping the most recent peers, as judged by the peers' associated timestamps, which are also sent over the network. It should be noted that Hlux checks if a peer received over the network does not have a timestamp in the future. If it does, then this peer is rejected. Without this check, it would be quite easy to permanently poison the peerlists of Hlux bots.

Vulnerabilities

In March 2012, a team from Kaspersky Labs was able to abuse Hlux's peerlist update mechanism to significantly poison the peerlists of all Hlux bots. This poisoning was facilitated by the fact that the Hlux protocol allows attackers to actively connect to any Hlux peer, and then push 250 poisoned peerlist entries to that peer. By setting a very recent timestamp for each pushed peer, it is possible to make the remote peer incorporate most if not all of the pushed peers into its peerlist, overwriting many legitimate entries in the process.

By pushing many peerlist entries into the Hlux network, all of which pointed to a rogue server operated by Kaspersky, the Kaspersky Labs team was able to severely disrupt the normal p2p communications of the Hlux bots. Legitimate Hlux bots were not connected with each other anymore, but were instead only connected to the rogue Kaspersky server.

At the same time, the Kaspersky team spread a list of false control server domains throughout the Hlux network, thereby preventing the Hlux bots from connecting to the real command and control servers to retrieve commands and updates.

Hlux's real control servers were subsequently disabled through legal action by Microsoft, crippling the Hlux botnet further [15]. This combination of poisoning and takedown of the botnet's backend control domains is reminiscent of the Waledac takedown.

Although the takedown was quite effective and succeeded in disabling the original Hlux botnet completely, the Hlux botmasters created a fresh Hlux botnet only months after the takedown of the original Hlux.

2.7 Miner

The Miner botnet is named after one of its main monetization techniques, which involves using the bots to generate large amounts of a digital currency called Bitcoin. This activity is known as Bitcoin mining, hence the name of the botnet. Apart from Bitcoin mining, the Miner botnet is also used for Denial of Service attacks, identity theft, and click fraud.

The Miner botnet appeared in August 2011, and contains 80.000 *externally reachable* peers.

An estimation often considered reasonable is that every 1 in 10 peers in a p2p network is externally reachable. This would put the total size of the Miner botnet at around 800.000 peers, but because Miner bots exchange only externally reachable peers, there is no way to be certain that this number is correct [16].

Communication protocol

The contents of this section are based on the Miner analysis by Werner [16].

Miner uses a custom text–based protocol over HTTP. The Miner protocol is quite simple, and is mainly meant for spreading new binaries to the bots. Miner messages are transmitted without any form of encryption.

Miner bots contain initial peerlists of around 2000 bootstrap peers. A Miner bot wishing to contact another peer first probes that peer on TCP port 62999 to verify that the remote peer is really part of the Miner botnet. All further communication takes place over HTTP on TCP port 8080.

A Miner bot wishing to download a piece of information from another bot will issue an HTTP `GET` request to that bot. There are a number of well known resource names which Miner bots use to access different pieces of information, such as the remote peer's peerlist or the remote peer's list of binaries.

In response to a peerlist request, a Miner bot returns a list of the IP addresses of 300 to 800 of its peers. Note that only IP addresses are returned, without any identifiers or ports. The latter is not needed, as Miner bots are always contacted on fixed ports.

Miner peerlists can be arbitrarily long, and contain only externally reachable peers. By requesting its public IP address from another bot, a Miner bot is able to learn whether or not it is externally reachable. If it is not, then it will never advertise itself to be added to another bot's peerlist.

Binaries distributed through the Miner network are named with increasing sequence numbers. This allows Miner bots to recognize which files in a remote peer's file list are new. If a Miner bot finds that a remote peer has a new binary available, it issues a `GET` request for the new binary to the remote peer. All binaries distributed through the Miner network are digitally signed to avoid the injection of rogue binaries into the network.

2.8 Zeus

Zeus is the most recent p2p botnet that we know of. It first appeared in October 2011, and seems to have developed from the leaked source code of a centralized variant of Zeus. We have reverse engineered the Zeus p2p communication protocol, and attempted a first takedown against the Zeus p2p botnet.

The results of our Zeus reverse engineering efforts are described in detail in Chapter 3, and the results of our takedown attempt against Zeus are documented in Chapter 4.

Chapter 3

A Detailed Analysis of Zeus

This chapter describes the results of our reverse engineering analysis of Zeus, a p2p botnet which first appeared in October 2011. Zeus is a trojan with the main purpose of stealing credentials from infected hosts. Before the p2p variant, there have been two major centralized variants of Zeus. These were sold in the underground community as kits to create customized botnets. The botnet binaries generated with these kits were typically distributed via e-mail campaigns and drive–by downloads.

In May 2011, the source code of the then most recent of these centralized Zeus variants was leaked. It appears that the p2p variant of Zeus was developed from this leaked source code. In the remainder of this chapter, we focus only on the p2p mutation of Zeus.

Our reverse engineering results are focused mainly on the p2p communication model of Zeus. Assembly listings detailing some of the Zeus behaviour discussed in this chapter are listed in Appendix B. Our reversing results are based on Zeus samples with the following MD5 hashes.

```
17e808d2eb19818ca21e3eeb8c556c34
8b3cda277fedf923a8ec03fc5da79fc0
8e5e837d2204e1bc6c242d7b74d9f3e9
```

3.1 Extracting a Zeus binary

Zeus binaries are spread in packed form, so that they can not be reversed directly. In order to analyze Zeus, it must first be extracted somehow. The approach we use to extract Zeus is based on the observation that Zeus injects itself into other running processes.

To extract a Zeus binary, we begin by running Zeus inside a virtual machine. Shortly after it is started, Zeus will inject itself into a running process, typically the Windows `explorer.exe` process. To tell into which process Zeus has injected itself, we wait until we see a process initiate suspicious network activity. In the samples we analyzed, the infected process opens a TCP socket and a UDP socket after about 5 seconds, and starts listening on these sockets. Once we know into which process Zeus has injected itself, we dump the memory of the virtual machine. We then analyze this memory dump using the Volatility Memory Forensics Platform[1].

We use the Volatility `malfind` plugin[2] to extract the actual Zeus binary from the full memory dump. The `malfind` plugin finds injected code inside a memory dump by looking for memory ranges which are marked executable, but are not listed in the Windows Process Environment Block (PEB). Because normally loaded executable code is always listed in the PEB, executable regions not listed there were likely injected by another process.

Using `malfind`, we look for an executable memory region starting with a Windows MZ header, which marks a Windows binary. In our memory dumps, the `explorer.exe` process contains exactly one such region. Dumping this region yields a Zeus binary suitable for analysis.

Detailed instructions on obtaining a Zeus binary can be found in Appendix A.

[1] `http://www.volatilesystems.com/default/volatility`

[2] `http://code.google.com/p/volatility/wiki/CommandReference#Malware_and_Rootkits`

```
int zeus_xor_encrypt(void *src, void *dest, int len) {
    if(src != dest) {
        memcpy(dest, src, len);
    }
    for(int i = 1; i < len; i++) {
        dest[i] ^= dest[i-1];
    }
    return len;
}
```

Figure 3.1: The Zeus network encryption algorithm.

3.2 Zeus communication model

This section describes our reverse engineering results on the Zeus p2p communication model. These results were obtained using a combination of network traffic analysis and static analysis in IDA Pro.

3.2.1 Overview

Zeus control messages are always sent over UDP, while TCP can be used for reliable file transfer. Zeus also supports file transmission over UDP. Zeus variants differ in which files they transmit over UDP, and which they transmit over TCP, but newer variants tend to prefer UDP rather than TCP in most cases. Details on the message types supported in Zeus can be found in Section 3.2.4 and Section 3.2.5.

Zeus bots come with a hardcoded initial peerlist. The samples we analyzed had initial peerlists containing 50 peers. All peers in the peerlist are periodically probed to confirm that they are still alive. If they are not, they are deleted. If their peerlists grow too small, Zeus bots may decide to ask other peers for additional peerlist entries. Zeus bots limit their peerlists to a maximum of 150 entries. Zeus stores its peerlist RC4 encrypted in the Windows registry under a pseudorandomly generated subkey of `HKEY_CURRENT_USER`.

All Zeus bots are able to periodically download updated binaries and configuration files from other peers. New binaries and configuration files are pushed into the Zeus network by the botmasters in order to update the bots and tweak their behaviour. New binaries contain updated Zeus versions, while configuration files are used to command the bots. Zeus bots do not receive commands via C&C messages, but only via new configuration files. Zeus bots store their configuration files encrypted using RC4, with a separate key per bot. An initial configuration file containing default settings is hardcoded into each Zeus bot.

Each peer in the Zeus network has an identifier in the form of a randomly generated SHA1 hash, and peers having similar identifiers are considered "close" to each other. The similarity between peer ID's is calculated using a Kademlia–like XOR–metric [9]. Zeus peers use their identifiers to find peers close to themselves, and to be able to recognize known peers even if they have dynamically changing IP addresses. Unlike Kademlia peers, Zeus bots use flat peerlists, and the Zeus architecture is *not* a Distributed Hash Table (DHT).

Periodically, some Zeus peers are designated as *proxies*. The botmasters push special signed packets into the network advertising these proxy peers. The proxy peers are then used by other Zeus bots to drop stolen data. It is possible that Zeus uses a multilayered architecture, and that the proxies forward dropped data to a higher layer, but we have not yet confirmed this.

Zeus uses a backup network based on a Domain Generation Algorithm (DGA). The domain generation algorithm generates pseudorandom domain names based on the current time and date, which Zeus bots can try to contact should they need to. The botmasters periodically register one or more of these domains, so that bots in DGA mode eventually successfully reach a domain.

Zeus bots activate their DGA when they find they have an empty peerlist, and thus can not join the Zeus p2p network. Zeus bots are able to use the DGA to obtain a fresh peerlist, so that they can rejoin the p2p network.

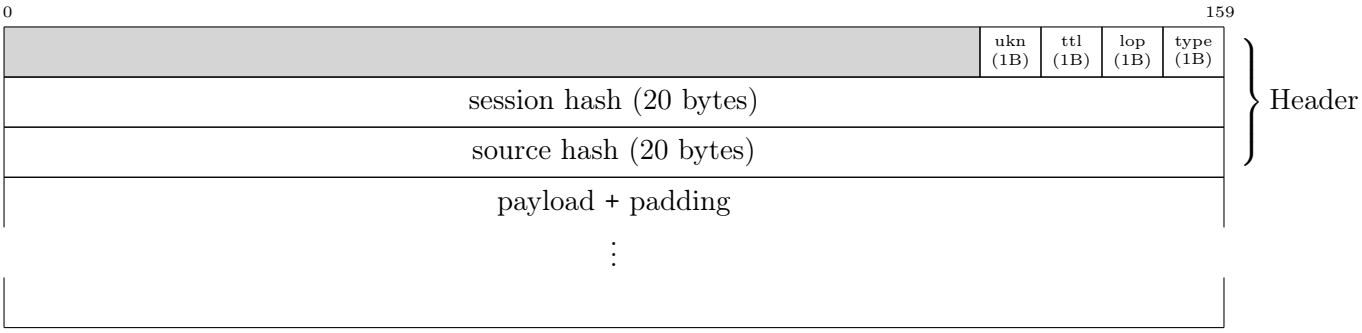


Figure 3.2: The basic Zeus network packet structure.

3.2.2 Encryption

Zeus has built–in support for various encryption algorithms, including RC4. RC4 encryption is used for encrypting sensitive data stored by Zeus bots, such as peerlists and configuration files. Additionally, Zeus bots use RC4 to encrypt binary and configuration file transfers.

Normal Zeus network traffic is only encrypted using a simple XOR encryption algorithm. The algorithm works by XORing each byte of a network packet with the preceding byte, starting at byte 1 (as byte 0 has no preceding byte). Decryption is identical, except that bytes are decrypted in the opposite order they are encrypted in, starting at the last ciphertext byte and moving down to byte 1.

A representation in C of the Zeus network encryption code is shown in Figure 3.1. An assembly listing showing the XOR algorithm as it is found in Zeus binaries can be found in Appendix B.

3.2.3 Packet structure

This section describes the basic structure of Zeus network messages. In general, Zeus UDP and TCP messages follow the same structure, with the exception of TCP file transmission packets (see Section 3.2.5).

Zeus packets vary in size, but have a minimum length of 44 bytes. The first 44 bytes of each packet form a custom header, while the remaining bytes form a payload concatenated with an amount of padding. The Zeus packet structure is illustrated in Figure 3.2. The shaded area at the beginning of the figure does not represent part of the Zeus packet structure. It is only used in the figure to align the fields appropriately. The meaning of each of the fields shown in Figure 3.2 is detailed below.

ukn (unknown)

The meaning of the first header byte is unknown. Static analysis shows that this byte is set to a random value by Zeus. This may simply be done to avoid leaking information, as the XOR encryption Zeus uses for network traffic leaves the first byte of each packet in plaintext (a result of the fact that the first byte has no previous byte to be XORed with).

ttl

The ttl field is usually unused, and thus set to a random or constant value by Zeus, depending on the specific Zeus variant. However, for some message types, this field serves as a Time To Live (TTL) value. To our knowledge, this is currently only true for messages of type `0x32`. See also Section 3.2.4.

lop (length of padding)

Zeus packets end with a random amount of padding. The length of padding (LOP) field indicates the number of padding bytes present at the end of the packet. Zeus bots use this field to determine the length of the message payload by subtracting the LOP and the 44 header bytes from the total packet length.

type

This field indicates the type of the message. The message type is used to determine the structure of the payload, and in certain cases the meaning of some of the header fields, such

as the ttl field. Valid Zeus message types, and their corresponding payload structures, are described in Section 3.2.4 and Section 3.2.5.

session hash

The session hash is a 20 byte random SHA1 identifier used by Zeus to determine with which request an incoming reply belongs. When a Zeus bot sends a request to another bot, it includes a random session hash in the request header. The corresponding reply will include the same session hash in the header, which is then used by the requesting bot to look up the original request. If a message with an unknown session hash arrives, it is discarded, making it difficult to spoof Zeus replies blindly. Messages with known session hashes but unexpected type numbers are also discarded.

source hash

This field contains the 20 byte identifier of the peer that sent the message. This field mainly serves for new bots to make themselves known in the network by pushing their identifier to other bots (see Section 3.2.6).

payload

This variable length field contains a message type dependent payload. The payload structures for all message types are described in Section 3.2.4 and Section 3.2.5.

padding

This field contains a random number of padding bytes. The exact number is specified in the padding length field of the message header. Each of the padding bytes is a nonzero randomly generated value. The padding field is postfixed to the message payload, and serves to confuse signature–based traffic analysis.

3.2.4 UDP message types

In this section, we discuss how each of the Zeus UDP message types is used, and how the corresponding payloads are structured. Zeus mainly uses UDP for control messages, but UDP data transfer is also possible. Recent Zeus variants prefer UDP data transfer over TCP data transfer for binary and configuration file downloads, but not for data drops. The reason for this may be that binary and configuration files are signed, while data drops are not. For data transfers where the signature can be used for verifying integrity, the use of TCP is redundant.

Version request (type `0x00`)

Version request messages are used to request the current binary and configuration file version numbers of other Zeus peers. Version request messages are sent to determine whether or not new binary or configuration file updates are available.

Version requests usually have no payload. However, sometimes they contain an 8 byte payload of the form `0x01000000RRRRRRRR`, where each `RR` represents a random byte and `0x01000000` is a little endian integer containing the value 1. This payload serves as a marker which any requesting peer can piggyback with a version request to indicate that it would like to receive a type `0x06` proxy reply message (see Section 3.2.4).

Version reply (type `0x01`)

A version reply contains the version numbers of the binary and configuration files that the responding peer currently has. The binary version number indicates which version of Zeus the peer is running, while the configuration file version number indicates which Zeus configuration file the peer has. A TCP port number is also sent to indicate on which port the responding peer can be reached to download the files via TCP. Version replies end with 12 random bytes. The reply structure is shown in Figure 3.3.

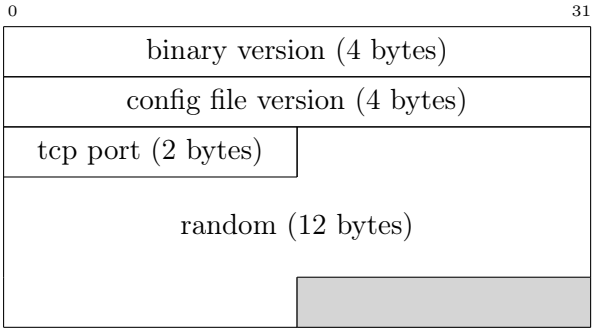


Figure 3.3: Version reply payload (22 bytes).

Peerlist request (type 0x02)

Peerlist requests are used to request new peers from other bots. The requesting peer adds these new peers to its peerlist, or updates the IP addresses and ports of peers with already known identifiers. Zeus peers do not usually use peerlist requests to learn about new peers. Instead, they usually obtain new peers by storing the senders of incoming request messages. Zeus only sends active peerlist requests if its peerlist is becoming critically short (less than 25 peers in the samples we analyzed).

The payload of a peerlist request consists of a 20 byte identifier, followed by 8 random bytes. The responding peer will return the peers it knows that are closest to the requested identifier. Zeus peers typically send peerlist requests containing the identifier of the peer they are sending the request to. The Zeus peerlist request structure is illustrated in Figure 3.4.

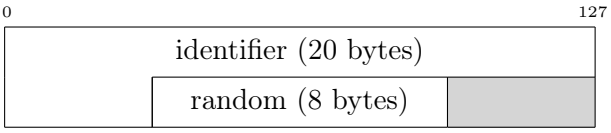


Figure 3.4: Peerlist request payload (28 bytes).

Peerlist reply (type 0x03)

Peerlist replies contain the 10 peers from the responding peer's peerlist which are closest to the requested identifier from the corresponding peerlist request. If the responding peer knows less than 10 peers, then as many peers as possible are returned. Peerlist replies containing no peers are also valid.

The payload length for a peerlist reply is always 450 bytes, large enough to contain exactly 10 peerlist entries concatenated together. If fewer than 10 peers are returned, the remaining space is padded with null bytes. The responding peer will never include itself in the peerlist it returns.

For each returned peer, the payload format is as shown in Figure 3.5. The ip type field indicates whether the peer is reachable via IPv4 or IPv6. A value of 0 for the ip type field indicates IPv4, while 2 indicates IPv6. The peer id field contains the identifier of the peer being returned, and the remaining fields contain the IP address and port where the peer can be reached via UDP. If IPv4 is used, the IPv6 fields are randomized. Similarly, if IPv6 is used, the IPv4 address is randomized. An interesting detail is that the IPv4 port is *not* randomized, but only set to zero. This may be a bug in Zeus.

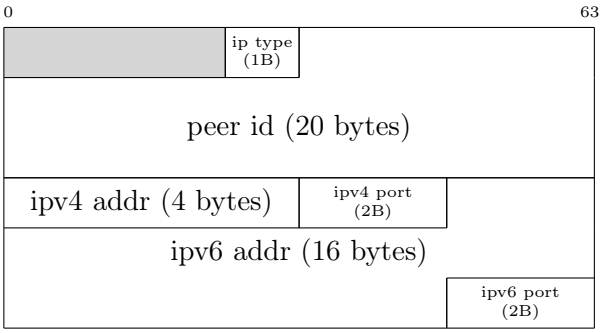


Figure 3.5: Peer struct (45 bytes).

Data request (type 0x04)

UDP data request messages are used to request binary or configuration file downloads via UDP. The Zeus UDP data request message structure is shown in Figure 3.6.

The payload of a UDP data request starts with a single byte indicating what kind of data is desired. This byte is set to 1 for a configuration file download, or to 2 for a binary update. The offset field indicates at which byte the responding peer should start transmitting data, and the size field specifies how many data bytes should be transmitted in the response. The size field is typically set to 1360 bytes. For large downloads, it is typical to see multiple type 0x04/0x05 messages exchanged in sequence.

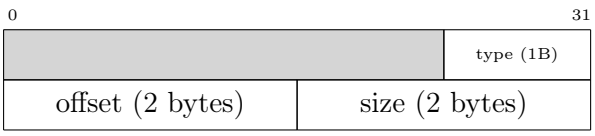


Figure 3.6: Data request payload (5 bytes).

Data reply (type 0x05)

UDP data replies contain the data requested by UDP data requests. UDP data replies always contain 1360 data bytes, except if there is no more data available. If a Zeus peer downloading

a file receives a data reply containing less than 1360 data bytes, it assumes that this is the last data block of the file, and ends the download. If a data reply takes longer than 5 seconds to arrive, the download is also aborted. The maximum total size of any download is 10MB. This limit is enforced by the receiver of the data.

Each UDP data reply starts with a 4 byte file identifier, for which any value is valid as long as all data replies belonging to the same file transmission use the same identifier. The file identifier is followed by the data requested in the previous data request packet. See also Figure 3.7.

The transmitted files end with a 256 byte RSA signature of the MD5 hash of the plaintext data, and are doubly encrypted with Zeus's XOR encryption algorithm, followed by an RC4 encryption layer using a hardcoded key. Before applying a downloaded binary or configuration file update, Zeus compares the version number contained in the update file with its current version number. If the new version number is not strictly higher than the current version number, the update is not applied. This means that it is not possible to make Zeus bots revert to old binary or configuration file versions by replaying old updates.

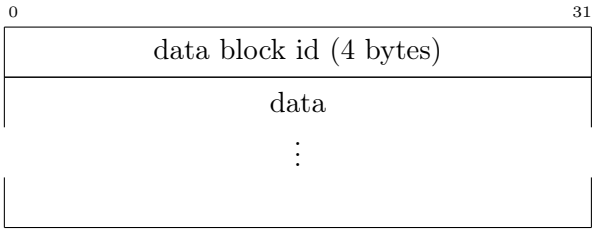


Figure 3.7: Data reply payload (length varies).

Proxy reply (type `0x06`)

Proxy replies are used to deliver the identifiers and addresses of peers where stolen data can be dropped. We refer to such peers as *proxies*. Proxy replies are sent in response to version requests with piggybacked proxy request markers. A proxy reply can contain up to 4 proxy entries, each of which is signed separately.

Each proxy entry in a proxy reply is formatted as shown in Figure 3.8. Proxy entries are formatted the same way as the peer structs used in peerlist replies, except that the ip type field is 4 bytes long instead of 1 byte, and there is a 256 byte RSA signature at the end of each proxy entry. The reason for the longer ip type field is unknown. Proxy reply entries do not contain timestamp fields, meaning that it is possible to replay old proxy reply entries.

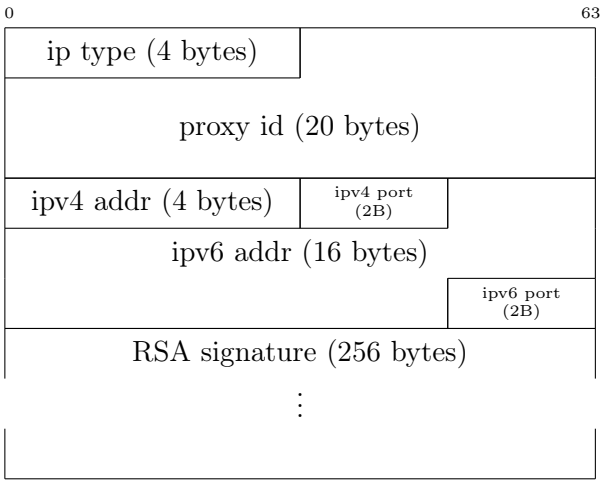


Figure 3.8: Proxy struct (304 bytes).

Proxy push message (type `0x32`)

Proxy push messages, like proxy replies, serve to inform Zeus peers about proxies where stolen data can be dropped. The difference is that proxy push messages are actively pushed through the Zeus network, rather than being sent in response to any message.

Proxy push messages utilize the ttl field in the Zeus header (see Section 3.2.3). The ttl field has an initial value of 4 for proxy push messages. Each Zeus peer which receives a proxy push message decrements the ttl field in the header, and then forwards the message to each peer in its peerlist. This way, proxy push messages propagate very rapidly through the Zeus p2p network.

Each proxy push message contains a single proxy entry of the same format used in type `0x06` messages, as shown in Figure 3.8.

3.2.5 TCP message types

Zeus uses TCP for two purposes. The first is to download binary and configuration files, although newer Zeus variants prefer to do this via UDP. The second is to connect to the proxies used for dropping stolen data.

Zeus TCP communication begins with a message containing a normal Zeus header, indicating what kind of data is to be transferred. This is followed by a download or upload of the actual data. The data is transferred in packets containing only the size of the data and the data itself, *without the usual Zeus header*. File transfers are acknowledged by sending a marker packet containing only the payload `0x01000000`, again without a header.

An interesting detail is that Zeus TCP messages do not contain the usual random padding seen in Zeus UDP messages. Additionally, the length of padding field in Zeus TCP headers is set to an invalid value, rather than being zeroed out as would be expected for messages without padding. It is possible that Zeus uses the length of padding field for a different purpose in TCP messages than in UDP messages.

Data drop request (type `0x66`)

We have observed that TCP type `0x66` packets are followed by large uploads from the Zeus peers which initiated the TCP communication. We therefore suspect that type `0x66` packets request permission to drop data. However, we have not yet reversed the format of the data upload that follows type `0x66` packets.

Binary request (type `0x68`)

TCP binary request packets are used to request updated Zeus binaries. Unlike UDP data requests, TCP binary requests do not specify which data offsets and sizes are desired, as TCP automatically takes care of these issues.

Configuration request (type `0x6A`)

TCP configuration requests are identical to TCP binary requests, except for the differing type number. Configuration requests are used to request new configuration files.

Data transfer

Zeus features a single TCP data transfer format used for every kind of data transfer. Data transfer packets do not carry Zeus headers, and thus do not have an associated type number. The format of TCP data transfer packets is extremely simple, as illustrated in Figure 3.9. It consists of 4 bytes indicating the size of the transmitted data, followed by the data itself.

Just as Zeus UDP downloads, binary and configuration file downloads via TCP end with a 256 byte RSA signature of the MD5 hash of the file being downloaded. Data transfers via TCP are RC4 encrypted using the session hash of the last exchanged UDP message as the key.

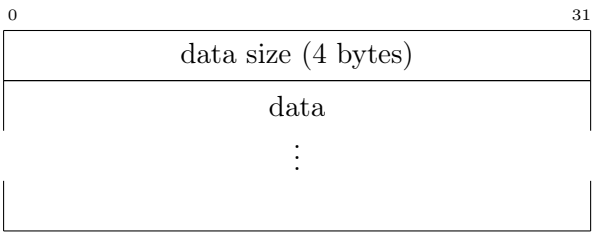


Figure 3.9: Data transfer (length varies).

3.2.6 Communication patterns

The previous sections have described the format and purpose of each of the Zeus message types. In this section, we discuss how all the message types fit together, and how Zeus bots use them to communicate.

Passive communication

Every Zeus bot listens for incoming messages from other bots. A Zeus bot receiving an incoming request will handle this request to the best of its abilities, and send back an appropriate reply, as described in Section 3.2.4 and Section 3.2.5.

The sender of any successfully handled request is considered for addition to the responding peer's peerlist. This is the main mechanism used by Zeus to learn about other peers, and it is also how new peers introduce themselves to the network.

If the responding peer currently knows fewer than 50 peers, then it always adds the sender of the request to its peerlist. Similarly, if the identifier of the sender is already present in the peerlist, then the corresponding IP address and port are updated. This is done because the sender may have a dynamically changing IP address.

If the identifier of the sender is not yet known to the responding peer, and the responding peer already knows at least 50 peers, then the sending peer's identifier, IP address and

port are added to a queue of peers to be considered for addition to the peerlist later.

Before adding a new peer to the peerlist, a number of sanity checks are performed. First, only peers which have a source port between 10000 and 30000 are added to the peerlist. Zeus bots always use ports in this range, though their externally visible ports may change in case they are behind a NAT. Zeus bots which are behind a NAT and which use external ports outside the normal Zeus range simply won't be present in any other peer's peerlist.

Additionally, there is a limit on the maximum number of peers with identical IP addresses in the peerlist. In the first Zeus variant we studied, this limit was set to 3, but it was changed to 2 in later variants. If adding a new peer to the peerlist would make its IP address too prevalent in the peerlist, then the new peer is discarded, but the existing entries with the same IP address remain.

Most incoming messages are requests from other peers, and are handled by sending back the appropriate reply type. However, type `0x32` proxy push messages propagate in a different way than normal Zeus messages, and deserve a little more attention here.

If a type `0x32` message arrives, it is first checked for validity. This is done using a number of checks on the message type number and the message length. Additionally, the included signature is checked for validity.

If the message passes the checks, the proxy it advertises is added to the receiving bot's list of proxies. The proxy list is similar to the peerlist, but is maintained separately. Zeus bots contact the peers listed in their proxy lists when they need to upload stolen data. It is worth noting that if a Zeus peer receives its own contact information in a proxy reply or proxy push message, it does not add this information to its proxy list.

If the identifier of a new proxy is already known in the proxy list, then the corresponding IP address and port are updated. Otherwise, if any of the proxies in the list is over 100 minutes older than the new proxy, then the first old proxy for which this is found to be the case is overwritten with the new proxy. In any other case, the new proxy is added to the end of the proxy list. Finally, the proxy list is truncated to its maximum length of 10 entries. Proxies received via type `0x06` messages are handled in the same way.

Next, if the message has a ttl value greater than zero, the ttl field is decremented. The type `0x32` message is then propagated to all peers in the peerlist.

Active communication

Besides passively listening for messages, all Zeus bots also actively participate in communication. The active communication of Zeus bots is illustrated in Figure 3.10, and described in detail in the remainder of this section.

The Zeus active communication pattern consists of a large loop which repeats every 30 minutes. The function of the active communication loop is to keep Zeus up to date, and to refresh the peerlist.

Zeus starts its active communication cycle by checking if it has any peers to communicate with. If not, it enters Domain Generation Algorithm (DGA) mode and attempts to retrieve a new peerlist from the domains generated by the DGA. If Zeus fails to reach any of the generated DGA domains, it keeps trying every 10 minutes until it succeeds.

After ensuring that it has peers to contact, Zeus proceeds to query each of its peers for their binary and configuration file versions. This step serves both to keep Zeus up to date, and to check each peer for responsiveness. Additionally, if Zeus currently knows fewer than 4 data drop proxies, it piggybacks a proxy request marker with each version request, causing the peers receiving the version requests to send back type `0x06` proxy reply messages in addition to their version replies.

Each peer is sent a version request a maximum of 5 times, with a 15 second timeout per request. If a peer fails to answer a request with a valid version reply, Zeus checks if it has working Internet access by attempting to contact `www.google.com` or `www.bing.com`. If it does, then the timeout is counted as a failure to respond by the remote peer. If Zeus does not have Internet access, it stops attempting to probe the current peer.

Peers which fail to respond to 5 subsequent version requests are deleted from the peerlist. If Zeus does receive a version response from a

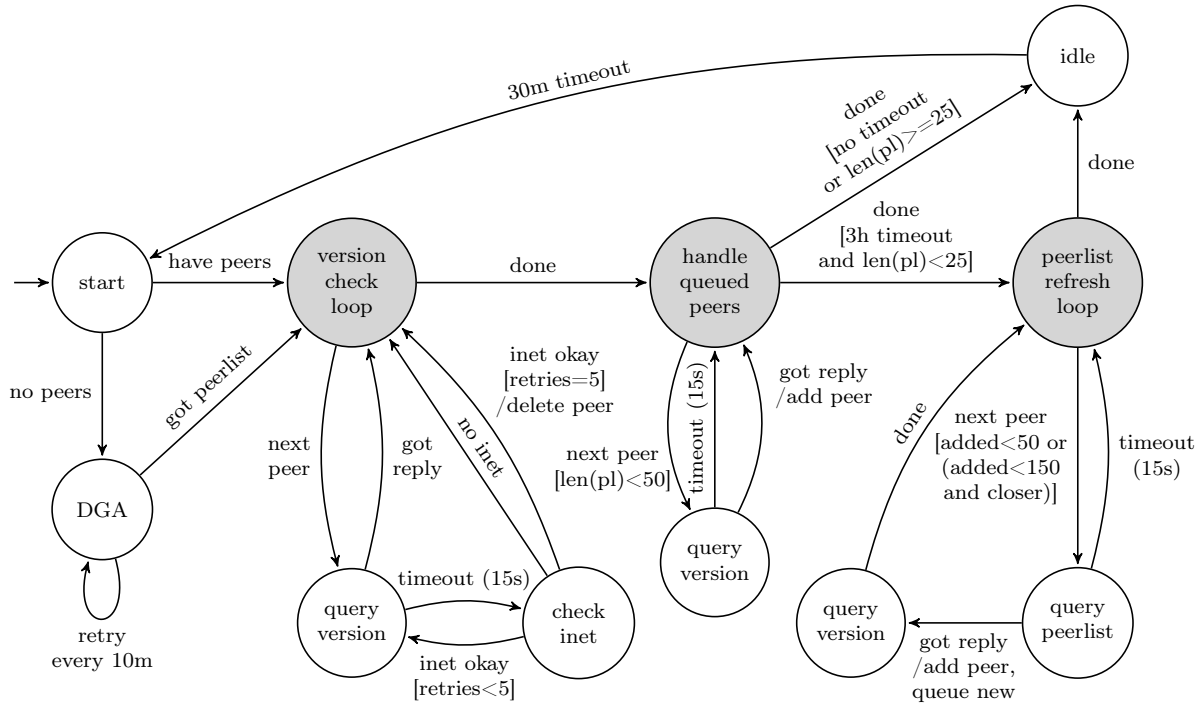


Figure 3.10: A state machine illustrating Zeus's active communication.

probed peer, it keeps this peer in its peerlist and checks if a new binary or configuration file version can be retrieved from this peer. The process of updating itself takes place outside of the main communication loop, so it is not shown in Figure 3.10.

After version querying all peers in its peerlist, Zeus proceeds to handle any pending peers which were queued from incoming requests (see Section 3.2.6). Each queued peer is handled as follows. If the peer's identifier is already known, then the corresponding IP address and port are updated as needed. Else, the pending peer is sent a single version request with a 15 second timeout. If the peer does not respond in time, it is discarded. If the peer does respond, then it is added to the peerlist. This process continues until the peerlist has length 50. After the peerlist has reached length 50, any still pending peers are left in the peer queue.

The next and final step in the loop is only executed once every 3 hours, and only if Zeus has a peerlist consisting of less than 25 peers. It can be considered an "emergency measure" to recover in case of a very small peerlist. This step, called the peerlist refresh cycle, is the only case where Zeus actively requests peerlists from other peers. Pseudocode describing the algorithm used in the peerlist refresh cycle is shown in Figure 3.11.

In the peerlist refresh cycle, Zeus constructs a completely new peerlist from its old peerlist and any still pending peers. A peer buffer is created which initially contains the entire old peerlist, and is then further filled with any peers which were still left pending after the previous active communication step.

Each peer from the peer buffer is considered for addition to the new peerlist. A peer from the peer buffer is added to the new peerlist either if the new peerlist is still shorter than 50 peers, or if the new peer is closer to the Zeus bot's own identifier than one of the other peers already in the new peerlist. This means that Zeus favors peers close to itself, and Zeus peerlists become biased towards the identifiers of the bots which own the peerlists.

Before a peer is added to the new peerlist, it is sent a peerlist request. If it does not respond, the peer is not added to the new peerlist, and Zeus continues with the next peer from the peer buffer. If the peer does respond, then all peers from the peerlist reply are queued in the peer buffer for possible addition to the new peerlist.

Each peer which successfully responded to the peerlist request is subsequently sent a ver-

```
new_peerlist = [] #array for the new peerlist
peer_buf     = [] #peers under consideration
peer_buf.append(registry.load_peerlist())
peer_buf.append(peer_queue)

for each peer p in peer_buf:
    peer_buf.remove(p)
    peer_queue.remove(p)
    if len(new_peerlist) >= 150:
        #maximum peerlist size is 150
        break
    if len(new_peerlist) < 50
      or (for some peer q in new_peerlist: dist(p,us) < dist(q,us)):
        #close peers are favored
        pl_reply = p.query_peerlist()
        if not pl_reply:
            continue
        new_peerlist.add_or_update(p)
        peer_buf.append(pl_reply)      #append all peers from pl_reply
        p.query_version()              #does not affect whether p is added

new_peerlist.order_by_timestamp()
registry.store_peerlist(new_peerlist) #replace peerlist with new_peerlist
```

Figure 3.11: The Zeus peerlist refresh algorithm.

sion request message. However, it is added to the new peerlist regardless of whether or not it responds to the version request. Version requests sent in this step always contain piggybacked proxy request markers. If the new peerlist reaches 150 peers, the process of adding new peers is stopped.

3.3 Crawling Zeus

Based on our reversing results, we have developed a Zeus crawler to estimate the size and global distribution of the Zeus p2p network. This section describes the design and results of our crawler.

3.3.1 Crawler design

Our crawler consists of three separate threads, namely a *sender thread*, a *receiver thread*, and a *logger thread*. The threads communicate with each other via two queues, which we call the *peer queue* and the *database queue*.

Initially, the peer queue contains a number of bootstrap peers. For early crawls, we manually selected 20 bootstrap peers, but after our initial crawls of the network we assembled a list of 30.000 bootstrap peers in order to speed up future crawls.

The peer queue is read by the sender thread. Each of the peers read from the peer queue is sent five peerlist requests. One of the requests contains the identifier of the queried peer, identical to peerlist requests sent by legitimate Zeus peers. The remaining four requests contain random identifiers to increase coverage of the network (recall that Zeus bots return peers close to the requested identifiers, so only requesting non–random identifiers yields poor coverage).

The sender thread keeps track of which peers it has already queried. To increase coverage of the network, the sender thread does not query the same peer twice until it has no other option. That is, it queries as many different peers as possible until the peer queue runs out, after which it pushes all of the explored peers back into the peer queue to be crawled again, with new random peerlist requests.

Any peerlist replies from the queried peers are read by the receiver thread. This thread parses the received peers from the peerlist replies and pushes them into the peer queue to be handled by the sender thread. Additionally, the peers are pushed into the database queue to be logged by the logger thread.

The logger thread logs each of the received peers. For each peer, we log a timestamp, the

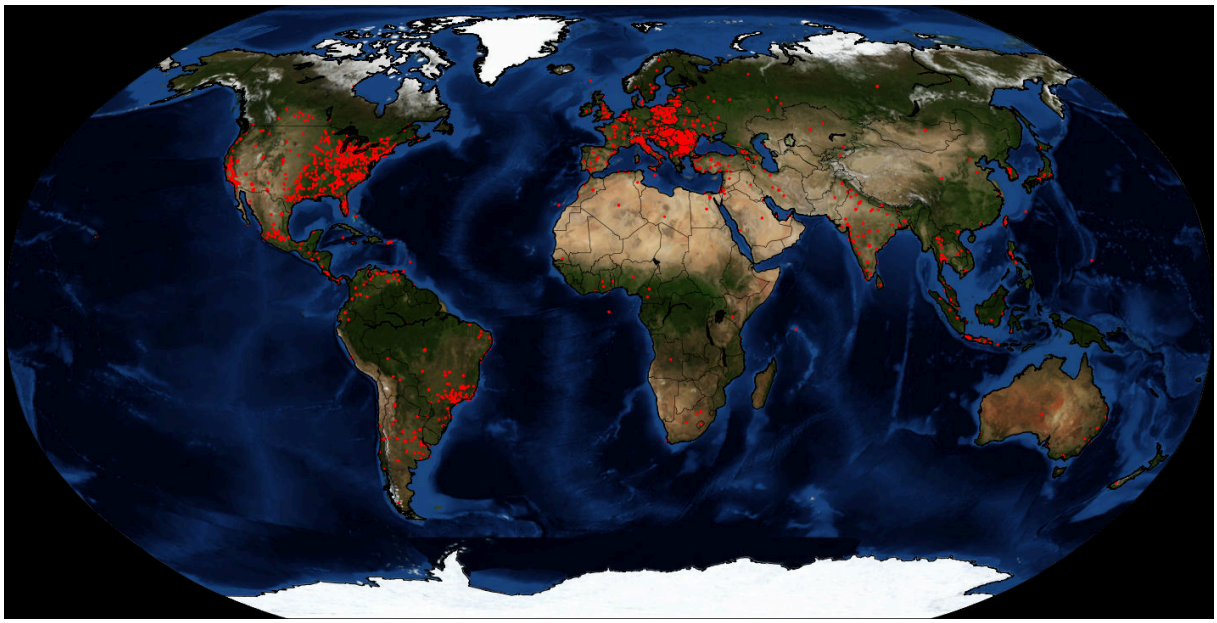


Figure 3.12: Geographic distribution of externally reachable Zeus peers.

identifier, IP address and port of the bot which sent us this peer, and the identifier, IP address and port of the peer itself. This information allows us to create connectivity graphs of the Zeus network, estimate the size of the Zeus network based on the number of unique identifers found, and determine the global distribution of the Zeus bots. Originally, the logger thread stored information directly into a PostgreSQL database. Because of performance issues, we later modified the crawler to log to comma–separated text files instead, which we periodically import into a database.

There is no real way to determine when a crawl of the Zeus network is "complete". Therefore, we simply let each of our crawls run for 24 hours before we manually terminate it. After 24 hours, very few new peers are still discovered, so we believe that we achieve reasonable coverage of the Zeus network. Letting the crawler run longer than 24 hours per crawl would pollute our results too much with dynamically changing IP addresses and churn from newly infected or disinfected bots.

3.3.2 Network size and distribution

Based on our crawling results, we estimate that the Zeus network currently contains between 150.000 and 200.000 bots. This estimate is based on the number of distinct peer identifiers found by our crawler in 24 hours.

We estimate that the Zeus network contains around 5.000 externally reachable peers. This number is determined by counting how many of the peers we found actually respond to our queries. NATed peers and peers behind firewalls typically do not respond to unexpected incoming UDP packets, meaning that they are not externally reachable.

Figure 3.12 shows the global distribution of the externally reachable Zeus peers found during one of our crawls. The locations shown are based on GeoIP results from the MaxMind GeoIP database[3].

[3] `http://www.maxmind.com`

Chapter 4

A Takedown Attempt Against Zeus

This chapter describes the results of our first takedown attempt against Zeus. Our takedown attempt took place during April and May of 2012, and was based on the reverse engineering results described in Chapter 3.

Although our takedown attempt did not disable the entire Zeus p2p botnet, we did manage to spread a significant amount of poison, reducing the botnet's ability to spread updates and data drop proxy locations. Additionally, we gained a better understanding of Zeus's behaviour under a poisoning attack. This improved understanding may prove useful in future takedown attempts.

4.1 Attack strategy

Recall from Section 3.2.6 that a Zeus bot receiving a request from another bot uses the source hash field of the request header to check if it already knows the requesting bot, and that if the requester's identifier is indeed already known, the receiving bot's peerlist is updated according to the source IP address and port of the request. Our attack is based on the fact that this behaviour can be used to actively poison the peerlists of Zeus bots.

4.1.1 Peerlist poisoning

The poisoning of a single Zeus peerlist entry is illustrated in Figure 4.1. In the example, the Zeus bot being poisoned initially has a peerlist with an entry for identifier `0xffff`, containing the associated IP address `198.51.100.49`. For clarity of the figure, 2 byte identifiers are used, and no port numbers are shown. In reality, Zeus keeps track of the 20 byte SHA1 identifier and last used IP address and port of each peer in its peerlist.

Next, a request is sent to the victim bot, with the source hash set to the identifier `0xffff` that is to be overwritten in our example. When the victim bot receives this request, it finds that it already has a peerlist entry with the identifier `0xffff`, and overwrites the address associated with this identifier with the source address of the received request.

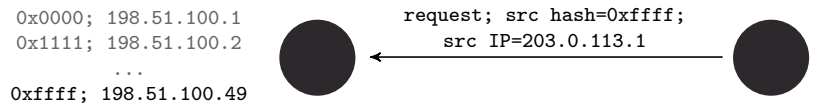
The source IP and port of a request can easily be spoofed, allowing an attacker to overwrite any known entry in the peerlist of a Zeus bot with any desired IP address and any port between 10000 and 30000 (the port range used by Zeus, see section 3.2.6).

4.1.2 Poisoning the Zeus botnet

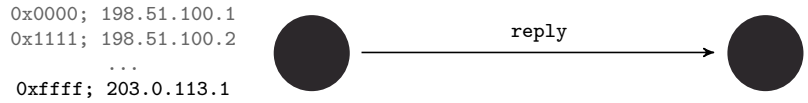
We now describe the full peerlist poisoning attack used during our takedown attempt against Zeus. Our attack aims to poison as many entries as possible in the peerlists of all Zeus bots so that these entries no longer point to legitimate Zeus peers. The eventual goal of the poisoning attack is to render the Zeus bots unable to update and receive commands.

In order to overwrite the peerlist of a Zeus bot, we first need to know which entries this peerlist contains. This is achieved by performing regular crawls of the Zeus network, as described in Section 3.3. By analyzing which peers are returned by each bot during a crawl, we obtain a partial view of each bot's peerlist.

Next, we attempt to overwrite each of the peerlist entries found during the most recent crawl, using the peerlist poisoning vulnerabil-



(a) Send a request to the victim with a crafted source hash.



(b) The victim overwrites its peerlist entry for the given source hash with the source address of the packet, then replies normally.

Figure 4.1: Poisoning a Zeus peerlist entry.

ity described in Section 4.1.1. We continuously repeat these crawling and poisoning steps. Because our crawler requests a number of random identifiers from each bot it contacts, we obtain a different partial view of each bot's peerlist during every iteration, increasing the coverage of our poisoning.

We use 60 different IP addresses to poison the peerlist entries of the Zeus bots. These IP addresses all point to a rogue Zeus server operated by us, which responds to any incoming request from a poisoned bot with a legitimate looking reply. This is done to ensure that we do not fail the periodic responsiveness checks performed by Zeus bots, as described in Section 3.2.6. Additionally, if an active peerlist request is received by our server, it responds with a list of peers crafted to poison as many of the requesting bot's peerlist entries as possible.

Our poisoned peerlist entries are spread further through the Zeus botnet by the bots themselves, as they communicate their peerlist entries to each other. Unfortunately, this effect is limited, since Zeus bots rarely actively request peerlist entries from each other (see Section 3.2.6). Normally, Zeus bots rely on finding other peers through the source hashes and source addresses of incoming requests, rather than through the exchange of possibly poisoned peerlist entries.

4.2 Takedown results

We started our takedown attempt on April 27th 2012. The poisoning program utilized during the takedown attempt uses 10 threads, each of which repeatedly retrieves the next node to poison from the list of nodes obtained during the most recent crawl, and then attempts to poison that node. During the takedown attempt, we ran the poisoning program roughly once a day. In the meantime, we relied on the Zeus bots to spread our poison amongst themselves, and poison themselves further by issueing peerlist requests to our rogue Zeus server.

To poison each node, the poisoning program utilizes a mix of tactics. First, it attempts to overwrite each of the already existing peerlist entries of the current poisoning target with one of our 60 IP addresses. The identifiers to overwrite for each target are determined from the results of the most recent crawl.

Additionally, the poisoning program attempts to extend the target bot's peerlist to a length of 50, to ensure that legitimate Zeus peers contacting the target after the poisoning attack are not immediately added to its peerlist (see also Section 3.2.6). This is done partially using random identifiers, and partially using identifiers close to the target's own identifier. The rationale behind the latter is that peerlist entries with identifiers close to the target are likely to be spread further when other bots request new peers from the target. This is true because the identifier requested in a peerlist request message is that of the receiver of the request (see Section 3.2.4).

On May 9th 2012, our poison had spread significantly through the Zeus p2p botnet, as shown in Figure 4.2. The x–axis of the figure shows the percentage of peerlist entries per bot pointing to one of our IPs, while the y–axis shows the percentage of bots for which this portion of their peerlists was poisoned. The figure is based on the results of our Zeus crawler.

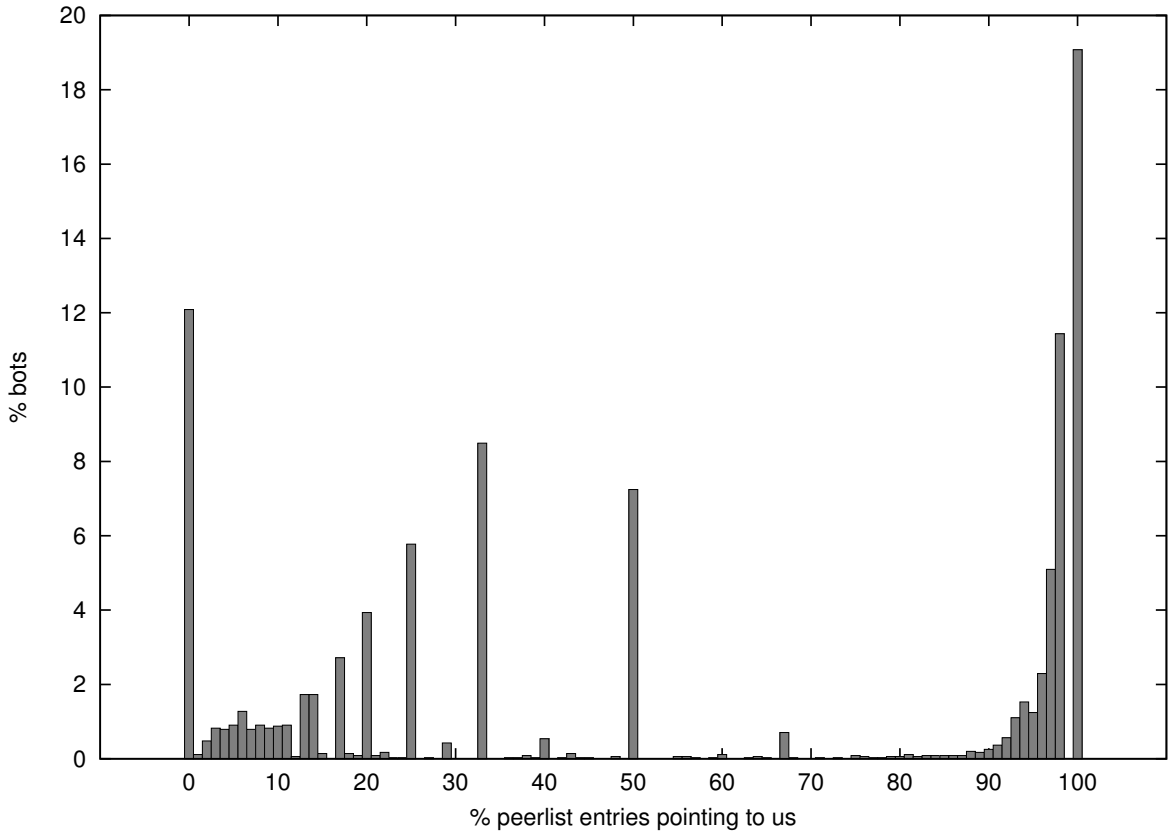


Figure 4.2: Percentage of peerlist entries pointing to one of our IPs versus percentage of bots for which this amount of peerlist entries was poisoned on May 9th 2012 (1 day before the Zeus IP blacklist update).

Figure 4.2 shows that on May 9th, only 12 percent of all bots contacted by our crawler did not return any peerlist entry pointing to one of our IPs. All other bots returned at least one entry pointing to our rogue Zeus server. Additionally, almost 20 percent of the bots returned exclusively peerlist entries pointing to our IPs, and about 44 percent of the bots returned one of our IPs in more than 90 percent of their peerlist entries. The average percentage of peerlist entries per bot pointing to one of our IPs was 54.83%.

On May 10th 2012, our takedown efforts were called to a halt by an update from the Zeus botmasters, in which they blacklisted the IP range we were using for our poisoning attack. Updated bots no longer accepted incoming messages from the blacklisted IPs, making it impossible for our rogue Zeus server to pass the periodic responsiveness checks performed by the Zeus bots. As a result, the updated bots quickly cleaned our IP addresses out of their peerlists, and regained connectivity to legitimate Zeus peers via the normal peerlist update mechanisms.

At the same time, the bots which were either completely poisoned or close to completely poisoned were unable to retrieve the blacklist update. Thus, these bots continued to contact our rogue server, and we were able to continue poisoning them, thereby preventing them from rejoining the normal Zeus network.

By May 19th 2012, 9 days after the botmasters blacklisted our IPs, only slightly more than half of all Zeus bots had received the latest update, as shown in Figure 4.3. Since Zeus bots are controlled via configuration file updates, bots which are unable to retrieve updates are rendered unusable to the botmasters. It is difficult to determine with certainty how many Zeus bots are unable to update due to our poisoning efforts, and how many are blocked from updating due to another factor. However, combined with the results from Figure 4.2, we estimate

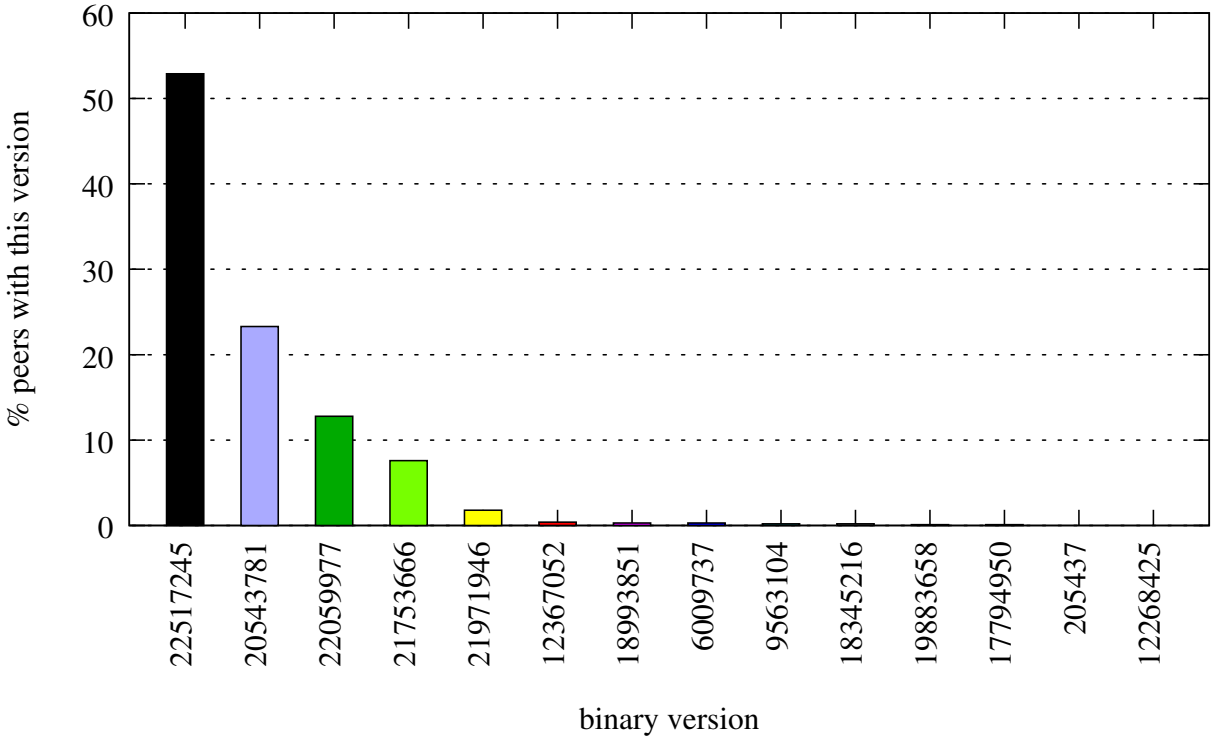


Figure 4.3: Zeus version distribution on May 19th 2012 (9 days after the Zeus IP blacklist update). Version `22517245` introduced the IP blacklist which blocked our poisoning IP range.

that at least the near 20 percent of completely poisoned peers are blocked from updating due to our efforts.

4.3 Discussion

Although our takedown attempt against Zeus did not succeed in disabling the entire botnet, we were able to inject a significant amount of poison into the botnet. On average, we managed to poison more than half of each bot's peerlist. A considerable fraction of the Zeus bots was rendered unable to update as a result of our poisoning efforts. Additionally, our takedown attempt has led to an improved understanding of how Zeus reacts to poisoning attacks, which may prove useful during future takedown attempts.

Perhaps the most important problem in our takedown attempt was that at the time of the takedown, we were not aware of just how rarely Zeus peers actively request peerlists from each other. As a result, we relied too much on the Zeus bots to spread our poison amongst themselves, while we should instead have been much more proactive in the injection of new poison into the network.

Furthermore, because Zeus peerlist requests are so uncommon, we received an unexpectedly low number of peerlist requests to our rogue Zeus server. Thus, we were unable to inject as much new poison into the Zeus network via peerlist replies from our rogue Zeus server as we had anticipated.

Since Zeus relies largely on incoming requests to learn about new peers, any poisoning effort against Zeus is a race against legitimate Zeus peers contacting each other. Thus, failing to inject new poison at a sufficient rate allowed the Zeus network too much time to recover.

Summarizing, we believe that a future takedown attempt against Zeus should aggressively inject poison into the Zeus network through use of the peerlist poisoning vulnerability described in Section 4.1.1, and should not rely on the Zeus bots to spread the poison amongst themselves. We expect that a sufficiently aggressive poisoning effort will result in a complete takedown of the Zeus p2p botnet.

Chapter 5

A Botnet Resilience Comparison

In the previous chapters, we have analyzed the designs and vulnerabilities of several major p2p botnets. This chapter provides an overview and comparison of the resilience of each of the botnets discussed in the previous chapters.

By comparing the characteristics of botnets which were successfully taken down to those of botnets which have survived exceptionally long, we derive a number of characteristics which contribute to botnet resilience. Furthermore, we estimate the resilience of each of the compared botnets to peerlist poisoning, one of the most promising and generalized attack vectors against fully decentralized p2p botnets.

5.1 Botnet survivability

In this section, we analyze the life span of each of the discussed botnets, and compare the characteristics of botnets with short life spans to those of botnets with long life spans. From this, we determine which characteristics contribute most to p2p botnet resilience.

5.1.1 Botnet life spans

Figure 5.1 shows the life span of each of the botnets we have analyzed. Storm, Waledac and Hlux each lived for 1.5 to 2 years. Although Storm was never truly disabled, it had inherent vulnerabilities which allowed attackers to disrupt its command and control [8]. Waledac was introduced by the Storm authors about 1.5 years after the appearance of Storm, to address these vulnerabilities [10, 11]. After the introduction of Waledac, remnants of Storm remained alive for about half a year. About 2 years after its introduction, Waledac was also taken down, and was succeeded by Hlux after several months. Hlux, in turn, was also disabled roughly 1.5 years after it first appeared, but it should be noted that Hlux has since been restarted[1], and is now significantly larger than it was in its first iteration [15].

The other botnets shown in the figure have not yet been disabled. In the case of Miner and Zeus this can be explained by the fact that these botnets are quite new, and they are still being actively researched. While the Conficker C botnet has also not been disabled since its introduction in early 2009, and is among the longer living botnets, it should be noted that Conficker C appears to be in the process of dying out. It currently consists of only 25.000 bots, a fraction of the 200.000 bots it contained at its peak[2].

The two longest living p2p botnets are ZeroAccess and Sality. Of these two, Sality clearly stands out. It has been active since January 2008, and at the time of writing still continues its operations undisturbed [3].

5.1.2 Resilience factors

In Table 5.1, we compare the major characteristics of the discussed botnets. These characteristics are summarized from the botnet descriptions in Chapter 2.

Comparing the data from Table 5.1 with the botnet life spans shown in Figure 5.1, we observe a correlation between the life span of

[1] http://www.securelist.com/en/blog/655/Kelihos_Hlux_botnet_returns_with_new_techniques

[2] http://www.confickerworkinggroup.org/wiki/pmwiki.php/ANY/InfectionTracking#toc8

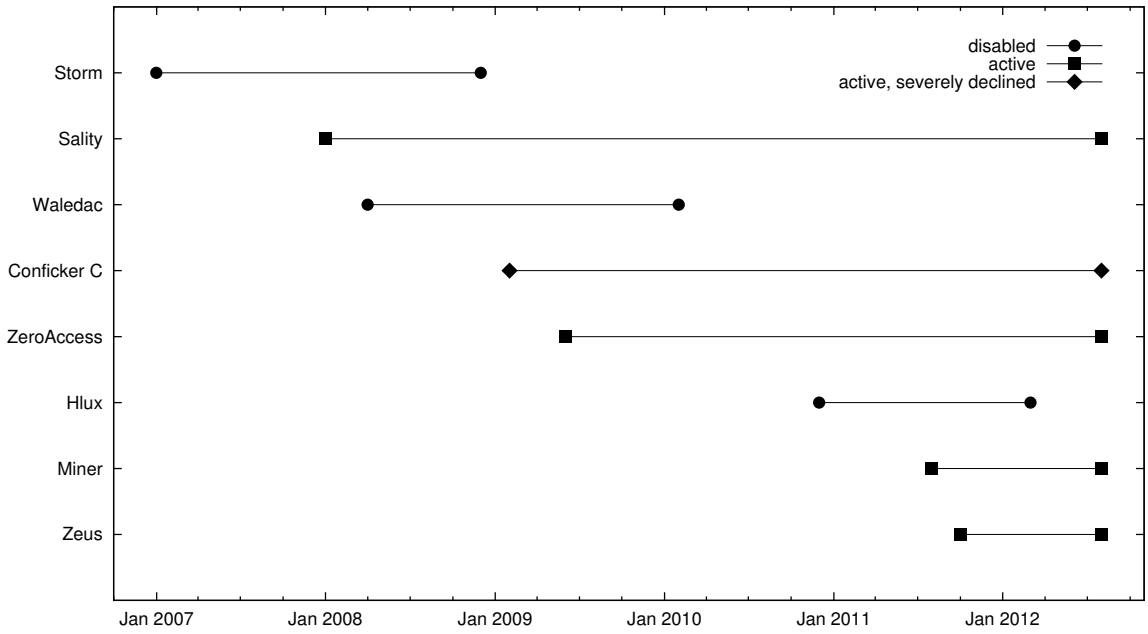


Figure 5.1: The lifetimes of the botnets discussed in this chapter.

Botnet	Ancestor	First seen	Last seen	#Bots ×1000	Protocol	Encryption	C&C signed	C&C routing
Storm		Jan 2007	Dec 2008	80	Overnet	XOR	No	Overnet lookups
Sality		Jan 2008		200	Custom	RC4	Yes	Gossiping
Waledac	Storm	Apr 2008	Feb 2010	165	Custom	AES	No	Router peers
Conficker C		Feb 2009		200 (25)	Custom	RC4	Yes	Gossiping
ZeroAccess		Jun 2009		150	Custom	RC4	Yes	Gossiping
Hlux	Waledac	Dec 2010	Mar 2012 (back up)	49 (130)	Custom	Blowfish + 3DES	Yes	Router
Miner		Aug 2011		800	Custom	None	Yes	Gossiping
Zeus		Oct 2011		200	Custom	Chained XOR	Yes	Gossiping

Table 5.1: Comparison of the major characteristics of the discussed botnets. Characteristics in parentheses represent significant changes from the original situations.

each botnet, and the way that it routes C&C traffic. Namely, all of the botnets which were taken down route commands in a structured way. In the case of Storm, commands are published in the Overnet DHT under predictable keys. Both Waledac and Hlux use externally reachable peers (referred to as *router peers* in the table) to route command and control traffic between the bots and an upper layer of centralized command servers. In contrast, none of the longer living botnets uses a structured C&C routing approach. Instead, they rely on gossiping to spread commands. This observation suggests that unstructured C&C routing is generally more resilient against takedown attempts than structured C&C routing.

Botnets such as Storm, which use existing DHT–based protocols like Overnet, are typically vulnerable to index poisoning attacks which prevent them from routing commands to their bots. This is a result of the fact that DHT–based protocols typically allow any peer

to publish content, meaning that if an attacker can predict a DHT–based botnet's command keys, it is possible to overwrite the botnet's commands as soon as they are published [17].

Botnets like Waledac and Hlux, which use centralized backend servers, are vulnerable to takedown of these backend servers. It should be noted, however, that it is possible in Waledac and Hlux to send new lists of command server domains to the bots in case the existing servers are disabled. This is why the takedowns against both Waledac and Hlux consisted of legal action against their backend domains, combined with peerlist poisoning to disrupt the propagation of new backend domain lists [12, 15].

Another important botnet resilience factor is command signing. If a botnet allows the use of unsigned commands, attackers can insert their own commands into the botnet, potentially disabling it entirely by propagating their own binary update. Table 5.1 shows that all of the discussed botnets, except Storm and Waledac, implement command signing.

It should be noted that DHT–based botnets which implement command signing are not necessarily immune to index poisoning. Although command signing prevents the insertion of rogue commands into a botnet, it does not prevent index poisoning attacks which simply corrupt existing commands.

Table 5.1 does not show a relationship between packet encryption and botnet life span. Although nearly all of the compared botnets do implement some form of packet encryption, this typically involves the use of hard-coded keys or weak XOR encryption algorithms. Thus, packet encryption typically seems to serve mostly as an obfuscation layer, and does not significantly increase the resilience of any of the compared botnets.

5.2 Poisoning resilience

In Section 5.1.2, we identified two important factors for the resilience of p2p botnets. First, command signing greatly increases botnet resilience by preventing the injection of rogue commands. Second, unstructured command routing mechanisms generally appear to be more resilient than structured command routing mechanisms.

As shown in Table 5.1, most of the compared p2p botnets implement both command signing and unstructured command routing. Thus, most of these botnets are immune to rogue command injection and do not have any clear points of failure in their command routing infrastructures. Barring command overwriting attacks and attacks against weak command routing infrastructures, one of the most promising and generally applicable attacks against p2p botnets is peerlist poisoning. Therefore, we devote this section to studying the susceptibility of each of the discussed botnets to peerlist poisoning attacks.

In order to compare the susceptibility of each of the botnets to peerlist poisoning, we identify two factors which are crucial to the success of a peerlist poisoning attack. First, it must be possible to insert poisoned peerlist entries into the victim botnet at a sufficient *rate*. If this is not possible, then the poisoning attack will be diminished by the normal exchange of peerlist entries between legitimate bots. Second, bots must place a sufficient level of *trust* in new peerlist entries. Otherwise, if the victim botnet uses strong verification mechanisms for new peerlist entries, poisoned peerlist entries may be detected and rejected by the botnet.

The rest of this section analyzes these two factors in each of the compared botnets, in order to estimate how vulnerable each botnet is to peerlist poisoning.

5.2.1 Peer exchange rate

Table 5.2 compares several factors influencing the rate at which each of the compared botnets can be poisoned.

The Sality botnet immediately stands out. It has a fairly large maximum peerlist length of 1000, while only allowing a single peer to be exchanged at once. To further slow down passive peerlist poisoning, Sality only requests new peers every 40 minutes, and only if it currently has less than 980 peers in its peerlist. Although it is possible to actively request addition to a Sality bot's peerlist, Sality verifies each new peer by probing it, and does not allow duplicate IP addresses in its peerlist [3]. Thus, to fully occupy a Sality peerlist through active peer sharing, an attacker needs 1000 externally reachable IP addresses, or must share loopback

Botnet	Peerlist size (max)	#Peers per exchange (max)	Peer request frequency	Active peer sharing
Storm	128 buckets ×20 peers	20 (`CONNECT`)	Every 30 seconds	Yes (`PUBLICIZE`)
Sality	1000	1	Every 40 minutes (if < 980 peers)	Yes
Waledac	1000	200	Unknown	Yes
Conficker C	2048	Unbounded	Probabilistic (during peer scan)	Yes
ZeroAccess	256	256	Every 20 minutes	No
Hlux	500	250	Unknown	Yes
Miner	Unbounded	800	Unknown	Yes
Zeus	150	10	Every 3 hours (if < 25 peers)	Yes

Table 5.2: Factors influencing the potential peerlist poisoning rate of each botnet.

Botnet	New peer verification	Periodic peer verification	Serving peer verification	Peers overwritable	Peer preference
Storm	Reject duplicate IP + port	None	None	Yes	Oldest alive
Sality	Probe actively pushed peer	Check sufficient goodcount every 40 minutes	Must have responded to `pack exchange`	No	High goodcount
Waledac	None	None	None	No	Newest
Conficker C	Check version of discovered peer	None	None	No	Newest
ZeroAccess	None	None	None	No	Newest
Hlux	Reject duplicate IP + port	None	None	Yes	Newest
Miner	None	None	None	No	None
Zeus	Bound duplicate IPs	Check responsive every 30 minutes	None	Yes	Close to own identifier

Table 5.3: Trust factors influencing the resilience of each botnet to peerlist poisoning.

addresses instead. Sality peerlist poisoning is complicated further by Sality's peerlist update policy, as discussed in Section 5.2.2.

In contrast, Table 5.2 shows that the Waledac and Hlux botnets facilitate especially fast peerlist poisoning. They allow attackers to actively push 200 and 250 peers at once, respectively. These are quite significant amounts compared to the respective maximum peerlist sizes of 1000 and 500 [11, 15]. The feasibility of poisoning Waledac and Hlux has been demonstrated in practice during the previously discussed takedowns against these botnets. Conficker C, which allows attackers to actively push an unbounded number of peerlist entries, also seems to facilitate rapid peerlist poisoning [13].

ZeroAccess, although it allows the exchange of a full peerlist of 256 entries at once, does not support the active pushing of peers, so that attackers wishing to poison a bot must wait for that bot to contact them. Once this occurs, however, the bot can be poisoned completely at once [14].

The Miner botnet is a special case. Although it allows the exchange of 800 peers at once, its maximum peerlist length is unbounded, meaning that although a Miner bot's peerlist can be filled with many polluted entries, it can never be "poisoned completely". Thus, the utility of the fast peer exchange rate allowed by Miner is limited. Further difficulties poisoning Miner are discussed in Section 5.2.2.

The remaining botnets allow moderate peerlist poisoning rates. The feasibility of poisoning them depends on the verification mechanisms and update policies they implement, as discussed in Section 5.2.2.

5.2.2 Trust factors

In Table 5.3, we evaluate the degree to which each of the botnets trusts new peerlist entries. The degree of trust placed in new peerlist entries is evaluated in terms of the botnets' verifications performed on peerlist entries, as well as the botnets' peerlist update policies.

Of the compared botnets, Sality and Zeus perform the most elaborate peer verifications. Both Sality and Zeus periodically clean up their peerlists, attempting to discard any bad peers which may have been added. Sality does this every 40 minutes by deleting peers with insufficient goodcount values, indicating that they have behaved erratically in the past. Zeus cleans up its peerlist every 30 minutes by contacting each of the peers in its peerlist, and deleting peers which do not respond properly to its queries.

In addition, Sality only adds peers which are trying to push themselves into its peerlist if they successfully respond to `pack exchange` queries. As Sality does not allow multiple peerlist entries with the same IP address, entirely poisoning a Sality peerlist would require 1000 publicly reachable IP addresses. Furthermore, Sality only discards peers with low goodcounts, and legitimate peers are expected to maintain high goodcounts. Thus, it is quite unlikely that an attacker would be able to fully occupy a Sality bot's peerlist, as this would require the bot to have discarded all of its legitimate peers [3].

The same effect is not achieved by the address checks implemented by Storm and Hlux, as they only prohibit duplicate IP/port pairs, meaning that only a single IP address is sufficient to poison their peerlists as long as each peerlist entry uses a different port. The bound on duplicate IP addresses implemented by Zeus also does not provide it with the same resilience against peerlist poisoning as Sality, because Zeus peerlists are limited to 150 entries, so that with a bound of 2 duplicates per IP address, 75 public IP addresses are enough to poison Zeus.

The Storm botnet is based on Overnet, which always prefers contact with old peers and, as long as they are responsive, never discards them in favor of new peerlist entries. This complicates the poisoning of Storm, because for a poisoned peerlist entry to overwrite an existing entry in a Storm peerlist, it is required that the existing entry is unresponsive at the time of poisoning [8, 9].

As already mentioned in section 5.2.1, the Miner botnet implements an atypical peerlist handling policy. As seen in Table 5.2, Miner peerlists do not have maximum lengths. Thus, Miner bots never discard any of their peers. Furthermore, Table 5.3 shows that Miner peerlist entries can not be overwritten, as they consist only of IP addresses without associated identifiers. Combined, these characteristics mean that it is not possible to fully poison the peerlists of Miner bots. However, since Miner bots select peers to contact randomly from their peerlists, inserting many poisoned entries into the peerlists of Miner bots reduces the possibility that these bots select legitimate Miner peers to contact. Thus, although a full peerlist poisoning attack against Miner seems infeasible, it is feasible to significantly reduce the effectiveness of Miner bots [16].

The remaining botnets do not implement significant peer verification mechanisms. Moreover, Waledac, Conficker C, ZeroAccess and Hlux all implement policies which discard existing peerlist entries in favor of newer peers. Thus, it is straightforward to force these botnets to discard all of their existing entries and replace them with newer ones. Additionally, all of these botnets allow the exchange of many peers at once, so that poison can be injected at such high rates that the botnets are not expected to be able to recover [11, 13, 14, 15].

5.2.3 Summary

Our analyses from Section 5.2.1 and Section 5.2.2 are summarized in Table 5.4. The poisoning rate restrictions for the Miner botnet are rated weak, because Miner allows a high peer exchange rate. Nevertheless, we stress again that Miner peerlists are unbounded, and can thus never be "poisoned completely".

The results from Table 5.4 are mixed. Four of the analyzed botnets, namely Waledac, Con-

Botnet	Poisoning rate restrictions	Trust exploitation defenses	Overall poisoning resilience
Storm	Moderate	Moderate	Moderate
Sality	Strong	Strong	High
Waledac	Weak	Weak	Low
Conficker C	Weak	Weak	Low
ZeroAccess	Weak	Weak	Low
Hlux	Weak	Weak	Low
Miner	Weak	Strong	High
Zeus	Moderate	Moderate	Moderate

Table 5.4: Summary of peerlist poisoning resilience analyses from Section 5.2.1 and Section 5.2.2.

ficker C, ZeroAccess and Hlux, are rated low in terms of poisoning resilience, because they can be poisoned at high rates and implement little or no trust exploitation defenses.

At the same time, Sality seems to be very difficult to poison effectively, as it simultaneously implements strong trust exploitation defenses, and allows only a minimal peer exchange rate. Because the Miner botnet appears to be impossible to poison completely, its resilience against peerlist poisoning is also rated high. However, we add a side note that it is feasible to inject Miner bots with so many poisoned entries that their probabilities of contacting legitimate peers are severely reduced.

The remaining two botnets, Storm and Zeus, are rated moderately resilient against poisoning attacks. Poisoning attacks against these botnets appear quite feasible, but require more coordination than poisoning attacks against botnets like Waledac, Conficker C, ZeroAccess and Hlux.

Chapter 6

Related Work

In order to gauge the threat that may be expected from p2p botnets in the future, several researchers have designed their own hypothetical p2p botnets, and analyzed their strengths and weaknesses. Starnberger et al. [4] have designed Overbot, a hypothetical p2p botnet based on Kademlia. Overbot is designed to obscure the IP addresses of the bots in the network, so that the botnet's size and composition can not easily be estimated. In addition, Overbot's command routing protocol is designed such that it can not easily be disrupted by attackers who have captured some of the nodes in the network. Yan et al. [5] have introduced Antbot, a DHT–based tree structured p2p botnet which publishes commands under a different set of keys per tree level, so that an attacker poisoning some of the published commands does not automatically disrupt the entire botnet.

In [18], Dumitriu et al. discuss and analytically evaluate the resilience of p2p file sharing systems against Denial of Service attacks. File sharing DoS attacks are also relevant in the scope of p2p botnets, because they may in some cases be usable to disrupt a botnet's command and control infrastructure. Castro et al. [19] discuss techniques for secure routing in structured p2p overlay networks. Conceivably, the introduction of more secure structured p2p systems could lead to the development of a stronger generation of structured p2p botnets based on these systems.

Davis et al. [20] discuss the implementation of effective sybil attacks against p2p botnets. They evaluate the performance of several sybil–based attack scenarios via simulation. In [12], Sinclair et al. discuss the design of an attack against Waledac which uses a mix of peerlist poisoning and takedown of Waledac's centralized backend servers. This attack was used in practice to disable the Waledac botnet. Another attack which was implemented in practice is the takedown against Hlux. The details of this attack are discussed in [15].

Finally, several papers exist which systematically study and compare several p2p botnets, similarly to this thesis. In [21], Grizzard et al. provide an overview of p2p botnets, as well as a detailed case study on Storm. The overview is rather dated, but is still useful as a supplement to this thesis because it includes a number of very early p2p botnets which we did not discuss. In [1], Dittrich provides an overview of past botnet takedowns, including takedowns against several p2p botnets, and draws parallels between them. Leder et al. [22] provide an interesting study on past botnet takedowns, and the reasons for their success or failure. They argue that in order to remain combative against modern botnets, more offensive countermeasures are needed. Offensive countermeasures against botnets, such as disinfection of bots through remote exploitation, are currently complicated by legal issues. The paper's argument for more offensive countermeasures against botnets accentuates our conclusion that some current p2p botnets are highly resilient to conventional countermeasures like peerlist poisoning.

Chapter 7

Conclusions

We have analyzed and compared the resilience of several p2p botnets. In addition, we have reverse engineered and attempted a takedown against Zeus, the most recent p2p botnet that we are aware of.

Our reverse engineering analysis has shown that Zeus is a sophisticated botnet, and is certainly not trivial to attack. Our first takedown attempt against Zeus was only partially successful, and was discontinued due to a counter-attack by the botmasters after fully poisoning an estimated 20% of the Zeus bots. Nevertheless, we believe that a faster and more focused poisoning attack against Zeus may very well result in a full takedown of the Zeus p2p botnet.

Our comparison of the resilience of p2p botnets has shown that all of the discussed botnets currently alive implement command signing, so that they can not be disabled through the injection of rogue commands. Additionally, nearly all of the discussed p2p botnets which were not yet taken down use unstructured command routing, without any weak centralized components to attack. Peerlist poisoning appears to be the most promising generalized attack vector against these botnets. For this reason, we have estimated the peerlist poisoning resilience of each of the discussed botnets.

Our poisoning resilience comparison has shown that several current p2p botnets implement very weak defenses against peerlist poisoning, making them vulnerable to attack. On the other hand, the Sality botnet is exceptionally resilient against peerlist poisoning. It has survived since its introduction in early 2008, illustrating that correctly designed p2p botnets can be extremely difficult to disable.

Bibliography

[1] D. Dittrich, "So you want to take over a botnet," in *Proceedings of the 5th USENIX Conference on Large-Scale Exploits and Emergent Threats*, LEET '12, USENIX Association, 2012.

[2] B. Awerbuch and C. Scheideler, "Towards scalable and robust overlay networks," in *The 6th International Workshop on Peer-to-Peer Systems*, IPTPS '07, 2007.

[3] N. Falliere, "Sality: Story of a peer-to-peer viral network," technical report, Symantec Corporation, 2011.

[4] G. Starnberger, C. Kruegel, and E. Kirda, "Overbot: a botnet protocol based on Kademlia," in *Proceedings of the 4th International Conference on Security and Privacy in Communication Networks*, SecureComm '08, pp. 13:1–13:9, ACM, 2008.

[5] G. Yan, D. Ha, and S. Eidenbenz, "Antbot: Anti-pollution peer-to-peer botnets," *Computer Networks*, vol. 55, no. 8, pp. 1941–1956, 2011.

[6] P. Porras, H. Saïdi, and V. Yegneswaran, "A multi-perspective analysis of the Storm (Peacomm) worm," technical report, SRI International, 2007.

[7] R. Shullich, "Analysis of the Storm worm," technical report, City University of New York, 2008.

[8] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. Freiling, "Measurements and mitigation of peer-to-peer-based botnets: A case study on Storm worm," in *Proceedings of the 1st USENIX Workshop on Large-Scale Exploits and Emergent Threats*, LEET '08, pp. 9:1–9:9, USENIX Association, 2008.

[9] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the XOR metric," in *Revised Papers from the 1st International Workshop on Peer-to-Peer Systems*, IPTPS '01, pp. 53–65, Springer, 2002.

[10] B. Stock, J. Göbel, M. Engelberth, F. Freiling, and T. Holz, "Walowdac – analysis of a peer-to-peer botnet," in *Proceedings of the 2009 European Conference on Computer Network Defense*, EC2ND '09, pp. 13–20, IEEE, 2009.

[11] G. Tenebro, "W32.Waledac threat analysis," technical report, Symantec Corporation, 2009.

[12] G. Sinclair, C. Nunnery, and B. Kang, "The Waledac protocol: The how and why," technical report, Infrastructure Systems Research Lab/University of North Carolina, 2009.

[13] P. Porras, H. Saidi, and V. Yegneswaran, "Conficker C p2p protocol and implementation," technical report, SRI International, 2009. Available at `http://mtc.sri.com/Conficker/P2P`.

[14] J. Wyke, "Zeroaccess," technical report, Sophos Labs UK, 2012.

[15] T. Werner, "Botnet shutdown success story: How Kaspersky Lab disabled the Hlux/Kelihos botnet," technical report, Kaspersky Labs, 2011. Available at `http://www.securelist.com/en/blog/208193137/`.

[16] T. Werner, "The Miner botnet: Bitcoin mining goes peer-to-peer," technical report, Kaspersky Labs, 2011. Available at `http://www.securelist.com/en/blog/208193084/`.

[17] P. Wang, L. Wu, B. Aslam, and C. Zou, "A systematic study on peer-to-peer botnets," in *Proceedings of the 18th International Conference on Computer Communications and Networks*, ICCCN '09, pp. 1–8, IEEE, 2009.

[18] D. Dumitriu, E. Knightly, A. Kuzmanovic, I. Stoica, and W. Zwaenepoel, "Denial-of-service resilience in peer-to-peer file sharing systems," in *Proceedings of the 2005 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, pp. 38–49, ACM, 2005.

[19] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. Wallach, "Secure routing for structured peer-to-peer overlay networks," *SIGOPS Operating Systems Review*, vol. 36, pp. 299–314, 2002.

[20] C. Davis, J. Fernandez, and S. Neville, "Optimising sybil attacks against p2p-based botnets," in *Proceedings of the 4th International Conference on Malicious and Unwanted Software*, MALWARE '09, pp. 78–87, IEEE, 2009.

[21] J. Grizzard, V. Sharma, C. Nunnery, B. Kang, and D. Dagon, "Peer-to-peer botnets: Overview and case study," in *Proceedings of the 1st Conference on Hot Topics in Understanding Botnets*, HotBots '07, USENIX Association, 2007.

[22] F. Leder, T. Werner, and P. Martini, "Proactive botnet countermeasures – an offensive approach," in *Proceedings of the Conference on Cyber Warfare*, 2009.

Appendix A

Extracting Zeus with Volatility

This appendix details how to extract a Zeus binary from a memory dump using Volatility[1] and the `malfind` plugin for Volatility[2].

The first step is to run a Zeus sample in a Windows virtual machine. We use a Windows XP SP3 virtual machine running in VirtualBox 4.1.12 under Ubuntu 11.10. After a few seconds, Zeus opens up a number of network sockets in the process where it injected itself. In the samples we analyzed, one TCP socket and one UDP socket were opened in the `explorer.exe` process. The opening of sockets can be observed using for instance the Process Explorer program from Microsoft's SysInternals Suite[3].

Once Zeus is confirmed to be active, the next step is to dump the memory of the virtual machine. A number of approaches are possible to accomplish this. We use the Moonsols DumpIt utility[4], which should be run from inside the virtual machine. Zeus can then be extracted from the virtual machine memory dump using Volatility.

First, we confirm that Volatility correctly recognizes the memory dump.

```
$ volatility imageinfo -f zeus.raw
Volatile Systems Volatility Framework 2.0
          Suggested Profile(s) : WinXPSP3x86, WinXPSP2x86 (Instantiated with WinXPSP2x86)
                     AS Layer1 : JKIA32PagedMemory (Kernel AS)
                     AS Layer2 : FileAddressSpace (./memdumps/zeus.raw)
                      PAE type : No PAE
                           DTB : 0x39000
                          KDBG : 0x8054cde0
                          KPCR : 0xffdff000
            KUSER_SHARED_DATA : 0xffdf0000
           Image date and time : 2012-02-27 09:42:29
     Image local date and time : 2012-02-26 17:57:10
          Number of Processors : 1
                    Image Type : Service Pack 3
```

Once this has been confirmed, we inspect the list of running processes to find the ID of the process hosting Zeus (in our case this is `explorer.exe`).

```
$ volatility --profile=WinXPSP3x86 pstree -f zeus.raw
Volatile Systems Volatility Framework 2.0
Name                                          Pid    PPid   Thds   Hnds   Time
 0x82E68960:explorer.exe                     1280   1260     20    426 2012-02-27 09:30:09
. 0x82C4DDA0:VBoxTray.exe                    1992   1280      5     72 2012-02-27 09:30:14
. 0x82DD81D0:apimonitor-x86.                  652   1280     13    186 2012-02-27 09:31:35
.. 0x82DC64B0:8e5e837d2204e1b                1468    652      0 ------ 2012-02-27 09:33:05
```

[1] http://www.volatilesystems.com/default/volatility

[2] http://code.google.com/p/volatility/wiki/CommandReference#Malware_and_Rootkits

[3] http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx

[4] http://www.moonsols.com/windows-memory-toolkit

```
... 0x82DFB228:cmd.exe                      1052   1468      0 ------ 2012-02-27 09:33:10
... 0x82D7A140:soono.exe                    1516   1468      0 ------ 2012-02-27 09:33:07
. 0x82C5B020:DumpIt.exe                     1256   1280      1     40 2012-02-27 09:42:27
. 0x82C32020:NLClientApp.exe                2000   1280     10    285 2012-02-27 09:30:14
 0x82FC8A00:System                             4      0     56    183 1970-01-01 00:00:00
. 0x82D64020:smss.exe                        404      4      3     19 2012-02-27 09:30:00
.. 0x82E6A610:csrss.exe                      528    404     10    375 2012-02-27 09:30:03
.. 0x82D4B020:winlogon.exe                   556    404     16    491 2012-02-27 09:30:03
... 0x82D91020:services.exe                  604    556     15    252 2012-02-27 09:30:04
.... 0x82D276E8:svchost.exe                  908    604      9    240 2012-02-27 09:30:05
.... 0x82D4E980:svchost.exe                 1036    604     14    189 2012-02-27 09:30:07
..... 0x82C3AB88:wscntfy.exe                1920   1036      3     48 2012-02-27 09:30:13
.... 0x82C4EC60:alg.exe                     1872    604      6    101 2012-02-27 09:30:12
.... 0x82E76C28:spoolsv.exe                 1380    604     11    121 2012-02-27 09:30:10
.... 0x82D4D9A0:nlsvc.exe                   1508    604     10    153 2012-02-27 09:30:10
.... 0x82D473E0:svchost.exe                 1000    604      5     60 2012-02-27 09:30:06
.... 0x82D3FDA0:VBoxService.exe              764    604      7     94 2012-02-27 09:30:05
... 0x82D95020:lsass.exe                     616    556     19    334 2012-02-27 09:30:04
```

The process ID we need is listed under the `Pid` column. For `explorer.exe`, it is 1280. We now run malfind to extract Zeus from its host process.

```
$ volatility --profile=WinXPSP3x86 malfind -f zeus.raw -p 1280 -D hidden_dumps/
Volatile Systems Volatility Framework 2.0
Name                 Pid    Start      End        Tag      Hits   Protect
explorer.exe         1280   0x00fb0000 0xfecfff00 VadS     0      PAGE_EXECUTE_READWRITE
Dumped to: hidden_dumps/explorer.exe.2e68960.00fb0000-00fecfff.dmp
0x00fb0000   4d 5a 00 00 00 00 00 00 00 00 00 00 00 00 00 00    MZ..............
0x00fb0010   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x00fb0020   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x00fb0030   00 00 00 00 00 00 00 00 00 00 00 00 08 01 00 00    ................
0x00fb0040   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x00fb0050   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x00fb0060   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x00fb0070   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................


explorer.exe         1280   0x01580000 0x1580fff0 VadS     0      PAGE_EXECUTE_READWRITE
Dumped to: hidden_dumps/explorer.exe.2e68960.01580000-01580fff.dmp
0x01580000   b8 35 00 00 00 e9 8b d1 38 7b 68 6c 02 00 00 e9    .5......8{hl....
0x01580010   94 63 39 7b 8b ff 55 8b ec e9 a3 2e c9 75 8b ff    .c9{..U......u..
0x01580020   55 8b ec e9 7e 60 c4 75 8b ff 55 8b ec e9 94 e9    U...~`.u..U.....
0x01580030   c4 75 8b ff 55 8b ec e9 8a 2f c9 75 8b ff 55 8b    .u..U..../.u..U.
0x01580040   ec e9 4b 4d c4 75 8b ff 55 8b ec e9 9f 82 c4 75    ..KM.u..U......u
0x01580050   8b ff 55 8b ec e9 ab 90 c7 75 8b ff 55 8b ec e9    ..U......u..U...
0x01580060   98 89 c5 75 6a 2c 68 10 7b 1c 77 e9 59 79 c4 75    ...uj,h.{.w.Yy.u
0x01580070   8b ff 55 8b ec e9 de 9b c7 75 8b ff 55 8b ec e9    ..U......u..U...

Disassembly:
01580000: b835000000                    MOV EAX, 0x35
01580005: e98bd1387b                    JMP 0x7c90d195
0158000a: 686c020000                    PUSH DWORD 0x26c
0158000f: e99463397b                    JMP 0x7c9163a8
01580014: 8bff                          MOV EDI, EDI
01580016: 55                            PUSH EBP
01580017: 8bec                          MOV EBP, ESP
01580019: e9a32ec975                    JMP 0x77212ec1
0158001e: 8bff                          MOV EDI, EDI
01580020: 55                            PUSH EBP
```

The first injected region found by `malfind`, starting with an MZ header, contains the code we are looking for. A full dump of this region is created by `malfind` in the `hidden_dumps` folder. This dump can be imported into a disassembler such as IDA Pro for analysis.

Appendix B

Annotated Zeus Assembly Listings

This appendix contains several annotated assembly listings from the Zeus binaries we analyzed. These listings may help readers to begin reversing newly extracted Zeus binaries.

The following listing represents the XOR algorithm Zeus uses to decrypt network packets. The `src` pointer is passed on the stack, and points to the bytes to decrypt. The bytes are decrypted into the `dest` buffer, pointed to by `edx`. The `len` parameter contained in `eax` represents the number of bytes in the `src` buffer.

```
; int zeus_xor_decrypt(const void *src, const void *dest<edx>, int len<eax>)
cmp     [esp + src], edx                ; are src and dest arrays the same?
jz      short loop_preamble             ; if so, go straight to the loop
push    eax                             ; else push arguments...
push    [esp + src]
push    edx
call    custom_memcpy                   ; ...and call a custom memcpy
jmp     short loop_preamble

loop_main:
mov     cl, [eax + edx - 1]             ; load previous byte
xor     [eax + edx], cl                 ; and xor it with current byte

loop_preamble:
dec     eax
jnz     short loop_main

retn    4
```

Zeus also supports RC4 encryption. The RC4 encryption is used among other things to encrypt configuration files. The Zeus RC4 implementation is listed below. The parameter `S` points to the RC4 Sbox, which is extended to save the RC4 `i` and `j` parameters across encryption rounds. The `len`, `dest` and `src` parameters have the same meanings as in the XOR decryption algorithm listed above.

```
; int rc4_encrypt(char *S<eax>, int len<edx>, void *dest<ecx>, void *src)
push    ebp
mov     ebp, esp
push    ecx                             ; push dest pointer twice
push    ecx
push    edi                             ; save old edi
mov     edi, ecx                        ; copy dest pointer into edi
mov     cl, [eax + 100h]                ; i from previous round is stored at S + 256
mov     [ebp + i], cl
mov     cl, [eax + 101h]                ; j from previous round is stored at S + 257
mov     [ebp + j], cl
test    edx, edx                        ; test if there are bytes to encrypt
```

```
jz      short exit                      ; if not, jump to the exit point
mov     ecx, [ebp + rounds_left]
sub     ecx, edi
push    ebx
mov     [ebp + src_offset], ecx         ; src_offset = src_ptr - dest_ptr
mov     [ebp + rounds_left], edx        ; rounds_left = len
push    esi

loop:
inc     [ebp + i]                       ; i = i + 1
movzx   esi, [ebp + i]
mov     dl, [esi + eax]
add     [ebp + j], dl                   ; j = j + S[i]
movzx   ecx, [ebp + j]
mov     bl, [ecx + eax]
mov     [esi + eax], bl                 ; S[i] becomes S[j]
mov     [ecx + eax], dl                 ; S[j] becomes old S[i]
movzx   ecx, byte ptr [esi + eax]       ; ecx = S[i]
movzx   edx, dl                         ; edx = S[j] (old S[i])
add     ecx, edx                        ; add S[i] and S[j]...
mov     edx, [ebp + src_offset]
and     ecx, 0FFh                       ; ...mod 256
mov     cl, [ecx + eax]                 ; cl = S[(S[i] + S[j]) mod 256]
xor     cl, [edx + edi]                 ; xor keystream byte with plaintext byte
mov     [edi], cl                       ; save the output byte
inc     edi
dec     [ebp + rounds_left]
jnz     short loop

pop     esi
pop     ebx

exit:
mov     cl, [ebp + i]
mov     [eax + 100h], cl                ; save i in Sbox for next encryption round
mov     cl, [ebp + j]
mov     [eax + 101h], cl                ; save j in Sbox for next encryption round
pop     edi
leave
retn    4
```

The RC4 key scheduling algorithm is implemented as shown in the following listing. The parameter `S` again points to the RC4 Sbox, while the parameter `key` points to the key used to initialize the Sbox. The parameter `key_len` contains the length in bytes of the RC4 key.

```
; int rc4_key_schedule<eax>(char *S<eax>, char *key, int key_len)
push    ebp
mov     ebp, esp
push    ecx
push    ebx
xor     ecx, ecx                        ; i = 0
push    esi
push    edi
mov     [ebp + k], cl                   ; k = 0 (k is the key index, the same as i)
mov     [ebp + j], cl                   ; j = 0
mov     [eax + 100h], cx                ; S + 256/S + 257 store i/j, set both to 0
mov     esi, eax                        ; pointer to current S index (S[i])
mov     edx, 100h

initial_Sbox_loop:
mov     [esi], cl                       ; S[i] = i
inc     ecx                             ; i++
```

```
inc     esi                             ; increment pointer to current S index
cmp     cx, dx
jb      short initial_Sbox_loop         ; loop if i < 256

mov     esi, eax                        ; set esi to initial S index again
mov     edi, edx                        ; 256 rounds to go

key_scheduling_loop:
movzx   ecx, [ebp + k]
mov     ebx, [ebp + key]                ; base of key array
mov     cl, [ecx + ebx]                 ; cl = key[k] (analogous to key[i])
mov     dl, [esi]                       ; dl = S[i]
add     cl, dl                          ; cl = S[i] + key[i]
add     [ebp + j], cl                   ; j = j + S[i] + key[i]
movzx   ecx, [ebp + j]
mov     bl, [ecx + eax]                 ; bl = S[j]
inc     [ebp + k]                       ; k++
mov     [esi], bl                       ; S[i] = S[j]
mov     [ecx + eax], dl                 ; S[j] = dl, where dl contains old S[i]
movzx   ecx, [ebp + k]
cmp     ecx, [ebp + key_len]            ; if key index has hit key length...
jnz     short end_key_scheduling_round
mov     [ebp + k], 0                    ; ...then wrap it to 0

end_key_scheduling_round:
inc     esi                             ; set esi to point to next S index
dec     edi                             ; decrement number of rounds left
jnz     short key_scheduling_loop

pop     edi
pop     esi
pop     ebx
leave
retn    8
```

Zeus uses the following code to compute the payload length of a received message according to the formula `payload_len = packet_len - header_len - padding_len`. The `packet` parameter points to a struct containing all the information needed to handle the received message.

```
; int payload_len(zeus_packet *packet<ecx>)
movzx   eax, word ptr [ecx + 14h]       ; ecx + 14h points to msg len
cmp     eax, 44                         ; is the msg long enough?
ja      short valid_msg
xor     eax, eax                        ; if not, return zero
retn

valid_msg:
movzx   ecx, byte ptr [ecx + 18h]       ; ecx + 18h points to header[2]
add     eax, 0FFFFFFD4h                 ; -44 in two's complement
sub     eax, ecx                        ; subtract the value of header[2]
retn                                    ; payload_len = msg_len - 44 - header[2]
```

The XOR–metric used by Zeus to determine the distance between two peers is computed as follows. The `id1` and `id2` parameters contain the two identifiers for which the XOR–metric is to be computed. The XOR difference computed between the two identifiers is saved into the buffer pointed to by `metric_dest`.

```
; int calculate_xor_metric(char *id1<eax>, char *id2<ecx>, char *metric_dest<edx>)
push    esi
push    edi
```

```
mov     esi, edx                        ; esi indexes the metric dest buffer
push    14h                             ; decimal 20, number of bytes in SHA1 hash
sub     ecx, eax                        ; indexing offset to id2
sub     esi, eax                        ; indexing offset to metric
pop     edi                             ; size of SHA1 hash

loop_calculate_metric:
mov     dl, [ecx + eax]                 ; load byte from id2
xor     dl, [eax]                       ; xor with byte from id1
mov     [esi + eax], dl                 ; store byte in metric
inc     eax                             ; next byte
dec     edi                             ; decrement number of bytes left
jnz     short loop_calculate_metric

pop     edi
pop     esi
retn
```