An Analysis of the Zeus Peer-to-Peer Protocol

Dennis Andriesse and Herbert Bos

VU University Amsterdam, The Netherlands
{d.a.andriesse,h.j.bos}@vu.nl

Technical Report IR-CS-74, rev. April 10, 2014

Abstract

Zeus is a family of credential-stealing trojans which originally appeared in 2007. The first two variants of Zeus are based on centralized command servers. These command servers are now routinely tracked and blocked by the security community. In an apparent effort to withstand these routine countermeasures, the second version of Zeus was forked into a peer-to-peer variant in September 2011. This paper describes our insights into the topology and communication protocol of the peer-to-peer variant of Zeus.

1 Introduction

Since its first appearance in 2007, Zeus has grown into one of the most popular families of credential-stealing trojans. Due to its popularity, previous versions of Zeus have been extensively investigated by the security community [5, 12]. The internals of the first two versions of Zeus, which are based on centralized command and control (C2) servers, are well understood, and C2 servers used by these variants are routinely tracked and blocked [1].

In September 2011, the second centralized version of Zeus mutated into a peer-to-peer (P2P) variant, known as *P2P Zeus* or *Gameover*. Since P2P Zeus does not rely on centralized C2, it is immune to traditional countermeasures against Zeus.

Centralized Zeus variants are distributed as builder kits in the underground community, allowing each user to build a private Zeus botnet. Interestingly, this is no longer supported in P2P Zeus, which is based on a single coherent main P2P network. The main P2P network is divided into several virtual *sub-botnets* by a hardcoded sub-botnet identifier in each bot binary. While the Zeus P2P network is maintained and periodically updated as a whole, the sub-botnets are independently controlled to perform various malicious tasks. Our bot enumeration results indicate that the Zeus P2P network contains at least 200.000 bots [9]. The geographical distribution of the externally reachable peers is shown in Figure 1.

The Zeus P2P network serves two main purposes. (1) Bots exchange binary and configuration updates with each other. (2) Bots exchange lists of *proxy bots*, which are special bots where stolen data can be dropped and commands can be retrieved. Additionally, bots exchange neighbor lists (*peer lists*) with each other to maintain a coherent network. P2P Zeus also uses a Domain Name Generation Algorithm (DGA) [2] as a backup channel, in case contact with the regular P2P network is lost or is found to be non-functional.

This paper provides technical details on the topology and communication protocol of P2P Zeus. Our results are based on P2P Zeus variants we observed between February 2012 and April 2014. A comparison of the resilience of the Zeus P2P protocol to other P2P botnet protocols is provided in our earlier work [9]. The lifecycle of P2P Zeus has been analyzed by Stone-Gross [10]. Early insights on P2P Zeus were provided by `abuse.ch` [11] and Lelli [7], and more recent insights by CERT.pl [4].

The rest of this paper is organized as follows. In Section 2, we provide a description of the topology of the Zeus P2P network. Next, Section 3 describes the communication protocol in detail. Section 4 provides details on the Domain Generation Algorithm used as a backup channel by P2P Zeus. Finally, Section 5 concludes on our findings.

2 Network Topology

The Zeus network is organized into three disjoint layers, as shown in Figure 2. At the bottom of the hierarchy is the *P2P layer*, which contains the harvester bots. Periodically, a subset of the bots is assigned the status of *proxy bot* (shaded in Figure 2). This appears to be done manually by the botmas-

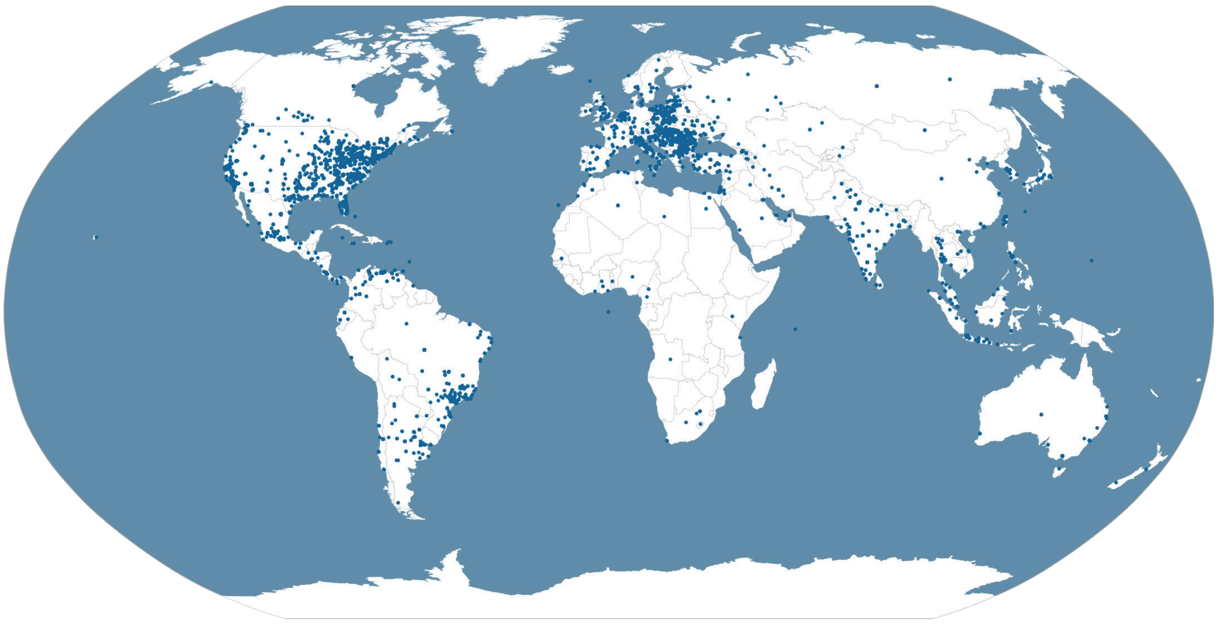

Figure 1: Geographical distribution of externally reachable Zeus peers.

ters, and is achieved by pushing a signed *proxy announcement* message into the network. The details of this mechanism are explained in Section 3. The proxy bots are used by harvester bots to fetch commands and drop stolen data. Apart from their role as proxies, proxy bots continue to exhibit the same behavior as harvester bots.

An important deviation from this behavior was introduced in some Zeus variants in March 2014. These variants use the Domain Generation Algorithm (Section 4) as the main C2 channel, and revert to the proxy bots only if the DGA cannot be reached. This behavior is not consistent as of April 2014; some live variants still use the proxy bots, while others default to the DGA. It is not yet clear which approach will become the default.

The proxy bots act as intermediaries between the P2P layer and a higher layer, which we call the *C2 proxy layer*. The C2 proxy layer contains several dedicated HTTP servers (not bots), which form an additional layer between the proxy bots and the true root of the C2 communication. Periodically, the proxy bots interact with the C2 proxy layer in order to update their command repository, and to forward the stolen data collected from the bots upward in the hierarchy.

Finally, at the top of the hierarchy is the *C2 layer*, which is the source of commands and the end destination of stolen data. Commands propagate downward from the C2 layer, through the C2 proxy layer to the proxy bots, where they are fetched by harvester bots. Similarly, data stolen by harvester bots is collected by the proxy bots, and periodically propagated up until it ultimately reaches the C2 layer.

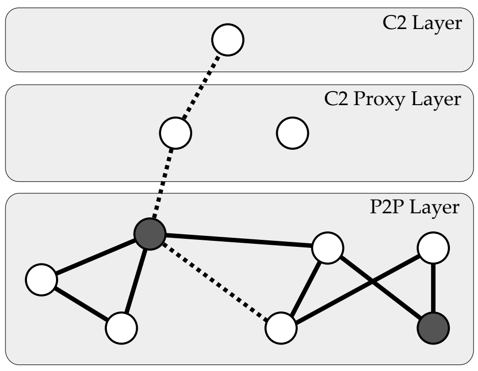

Figure 2: Topology of P2P Zeus. Proxy bots are shaded. The dotted line shows the information flow between a harvester bot and the C2 layer.

As mentioned in Section 1, the main P2P network is divided into several virtual *sub-botnets* by a hardcoded sub-botnet identifier in each bot binary. Since each of these sub-botnets is independently controlled, the C2 layer may contain multiple command sources and data sinks.

3 P2P Protocol

This section describes our analysis results on the Zeus P2P communication protocol. The results are based on Zeus variants we tracked between February 2012 and April 2014. In that time, several changes were made to the protocol by the Zeus authors. This section describes our current understanding of the protocol, based on the most recent variants we analyzed. However, note that results

```
; int zeus_xor_decrypt(void *src, void *dst<edx>, int len<eax>)
cmp     [esp + src], edx         ; are src and dst 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    memcpy                   ; ...and call 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
```

Figure 3: The deprecated Zeus rolling XOR decryption algorithm.

may differ for Zeus variants released after the revision date of this report.

We provide an overview of the Zeus P2P protocol in Section 3.1. Next, we describe the encryption used in Zeus traffic in Section 3.2. Sections 3.3 and 3.4 provide a detailed overview of the Zeus message structure. Finally, Section 3.5 describes the Zeus protocol state machine.

3.1 Overview

As mentioned in Section 1, the Zeus P2P network's main functions are (1) to facilitate the exchange of binary and configuration updates among bots, and (2) to propagate lists of *proxy bots*. Most normal communication between bots is based on UDP. The exceptions are C2 communication between bots and proxies, and binary/configuration update exchanges, both of which are TCP-based.

Bootstrapping onto the network is achieved through a hardcoded bootstrap peer list. This list contains the IP addresses, ports and unique identifiers of up to 50 Zeus bots. Zeus ports range from 1024 to 10000 for Zeus variants released after June 2013, and from 10000 to 30000 for older variants. Unique identifiers are 20 bytes long and are generated at infection time by taking a SHA-1 hash over the Windows *ComputerName* and the Volume ID of the first hard-drive. These unique identifiers are used to keep contact information for bots with dynamic IPs up-to-date.

Network coherence is maintained through a push-/pull-based peer list exchange mechanism. Zeus generally prefers to push peer list updates; when a bot receives a message from another bot, it adds this other bot to its local peer list if the list contains less than 50 peers. Bots in desperate need of new neighbors can also actively ask other bots for new peers.

Zeus bots check the responsiveness of their neighbors every 30 minutes. Each neighbor is contacted in turn, and given 5 opportunities to reply. If a neighbor does not reply within 5 retries, it is deemed unresponsive, and is discarded from the peer list. During this verification round, every neighbor is asked for its current binary and configuration file version numbers. If a neighbor has an update available, the probing bot spawns a new thread to download the update. Updates are signed using RSA-2048, and are only applied *after* the bot has checked that the update's embedded version number is higher than its current version. Thus, it is not possible to force bots to "update" to older versions.

The neighbor verification round is also used to pull peer list updates if necessary. If the probing bot's peer list contains less than 25 peers, it asks each of its neighbors for a list of new neighbors. The returned peer lists can contain up to 10 peers. The returned peers are selected by minimal Kademlia-like XOR distance to the requesting bot's identifier [8]. However, it is important to note that the Zeus P2P network is *not* a Distributed Hash Table, and apart from this XOR metric the protocol bears no resemblance to Kademlia.

In case a Zeus bot finds all of its neighbors to be unresponsive, it attempts to re-bootstrap onto the network by contacting the peers in its hardcoded peer list. If this also fails, the bot switches to a Domain Generation Algorithm (DGA) backup channel, which can be used to retrieve a fresh, RSA-2048 signed, peer list. Additionally, in recent variants of Zeus, the DGA channel is also contacted if a bot is unable to retrieve updates for a week or more. The DGA mechanism is described in more detail in Section 4.

As mentioned, one of the most important functions of the Zeus P2P network is to propagate lists of proxy bots. These proxy bots are a periodically selected subset of the general bot population, and are contacted by other bots to fetch commands

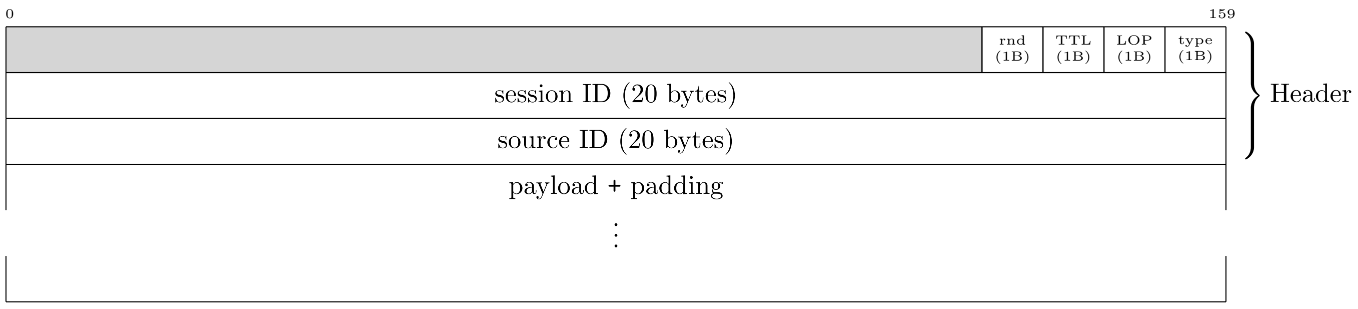

Figure 4: The Zeus message structure.

and drop stolen data. Like the peer list exchange mechanism, the proxy list propagation mechanism is also push-/pull-based. The botmasters appoint new proxy bots by pushing an RSA-2048 signed message, which is disseminated through the network using gossiping. This is the preferred proxy list update mechanism. However, bots in desperate need of a proxy list update can also actively request this from other bots.

Bots are commanded in two ways. (1) Bots can contact proxies to retrieve commands. (2) Configuration file updates can be used to effectively command the bots.

3.2 Encryption

Until recently, bot traffic was encrypted using a rolling XOR algorithm, known as “visual encryption” from centralized Zeus [12], which encrypts each byte by XORing it with the preceding byte (see Figure 3). Since June 2013, Zeus uses RC4 instead of the XOR algorithm, using the recipient’s bot identifier as the key. This new RC4 encryption is problematic for infiltration attempts which use randomized bot identifiers, as infiltrators must know under which identifier they are known to other bots in order to decrypt inbound traffic.

Zeus uses RSA-2048 to sign sensitive messages originating from the botmasters, such as updates and proxy announcements. Additionally, in all P2P Zeus variants we studied, update exchanges and C2 messages feature RC4 encryption over an XOR encryption layer. For these messages, the identifier of the receiving bot or a hardcoded value is used as the RC4 key, depending on the message type.

3.3 Message Structure

This section describes the structure of Zeus network messages. Zeus messages vary in size, but have a minimum length of 44 bytes. The first 44 bytes of each message form a header, while the remaining bytes form a payload concatenated with an amount of padding.

The Zeus message structure is illustrated in Figure 4. The shaded area at the beginning of the figure does not represent part of the Zeus message structure; it only serves to align the fields in the figure. The meaning of each of the fields shown in Figure 4 is explained below.

3.3.1 rnd (random)

In Zeus versions which use the XOR encryption, this byte is set to a random value. This is done to avoid leaking information, since the XOR encryption leaves the first byte in plaintext. In Zeus versions which use RC4 for message encryption, this byte is set to the bitwise inverse of the first session ID byte, so that it can be used to confirm that packet decryption was successful. Backward compatibility with older bots is achieved by falling back to the XOR encryption if RC4 decryption fails.

3.3.2 TTL (time to live)

The TTL field is usually unused, in which case it is set to a random value, or to the bitwise inverse of the second session ID byte for RC4-based variants. However, for certain message types, this field serves as a time to live counter. A bot receiving a message using the TTL field forwards it with a decremented TTL. This continues iteratively until the TTL reaches zero. For instance, this field is used in proxy announcement messages to prevent them from circulating forever.

3.3.3 LOP (length of padding)

Zeus messages end with a random amount of padding bytes. This is most likely done to confuse signature-based intrusion detection systems. The length of padding field indicates the number of padding bytes present at the end of a message. When receiving a message, a Zeus bot inspects the length of padding field, and then strips the padding bytes off the message.

3.3.4 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 are described in Section 3.4.

3.3.5 session ID

When a Zeus bot sends a request to a peer, it includes a random session ID in the request header. The corresponding reply includes the same session ID, and inbound replies with unexpected IDs are discarded. This makes it more difficult for attackers to blindly spoof Zeus messages.

3.3.6 source ID

This field contains the 20 byte bot identifier of the sending bot. The source ID field facilitates the push-based peer list update mechanism, where a bot receiving a message adds the sender of the message to its peer list in case the peer list contains less than 50 peers.

3.3.7 payload

This is a variable-length field which contains a payload dependent on the message type. The structures of relevant message payload types are described in detail in Section 3.4.

3.3.8 padding

This field contains a random number of padding bytes. The number of bytes contained in this field is specified in the length of padding field in the message header. Each of the padding bytes is a non-zero randomly generated value.

3.4 Payload Structure

In this section, we describe the structure and usage of the most relevant Zeus message types. Each of these message types is communicated over UDP, except for C2 messages and updates, which are exchanged over a TCP connection.

3.4.1 Version request (type `0x00`)

Version request messages are used to request a bot's current binary and configuration file version numbers. These messages typically contain no payload. However, an optional payload containing a little endian integer 1 may be present, followed by 4 random bytes. Such a payload serves as a marker to indicate that the requesting peer's proxy list is too short, and it wants to receive a type `0x06` proxy reply message (see Section 3.4.7).

3.4.2 Version reply (type `0x01`)

A version reply contains the version numbers of the binary and configuration files that the sender has. The binary version indicates which version of Zeus the peer is running, while the configuration file version indicates which Zeus configuration file the peer has. A TCP port number is also sent. This port can be contacted to download the updates via TCP, although some Zeus variants also support using UDP data exchanges for this (see Sections 3.4.5 and 3.4.6). Version replies end with 12 random bytes. The reply structure is shown in Figure 5.

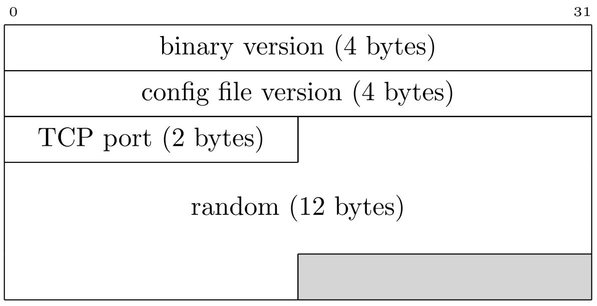

Figure 5: Version reply payload (22 bytes).

3.4.3 Peer list request (type `0x02`)

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

The payload of a peer list request consists of a 20 byte identifier, followed by 8 random bytes. The responding peer will return the peers from its own peer list that are closest to the requested identifier. The Zeus peer list request structure is illustrated in Figure 6.

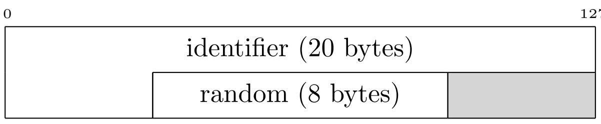

Figure 6: Peer list request payload (28 bytes).

3.4.4 Peer list reply (type `0x03`)

Peer list replies contain 10 peers from the responding peer's peer list which are closest to the requested identifier. If the responding peer knows fewer than 10 peers, then as many as possible (potentially zero) are returned. The payload length for a peer list reply is always 450 bytes, large enough for 10 peer list entries. If fewer than 10 peers are returned, the remaining space is null-padded.

For each returned peer, the payload format is as shown in Figure 7. The IP type field indicates whether the peer is reachable via IPv4 or IPv6. A value of 0 indicates IPv4, while 2 indicates IPv6. The peer ID field contains the identifier of the peer, and the remaining fields contain the IP address and UDP port. If IPv4 is used, the IPv6 fields are randomized. Similarly, if IPv6 is used, the IPv4 address is randomized. Curiously, the IPv4 port is *not* randomized, but only set to zero.

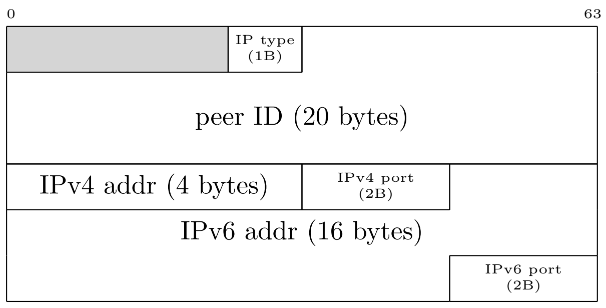

Figure 7: Peer struct (45 bytes).

3.4.5 Data request (type `0x04`/`0x68`/`0x6A`)

UDP data requests (type `0x04`) are used to request binary or configuration updates via UDP. The message structure is shown in Figure 8.

The payload of a UDP data request starts with a byte indicating the kind of desired data. This byte is set to 1 for a configuration file download, or to 2 for a binary update. The offset field indicates at which word 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.

TCP data requests consist of a message header with type `0x68` for a binary request, or type `0x6A` for a configuration request. The request is RC4 encrypted with the identifier of the recipient.

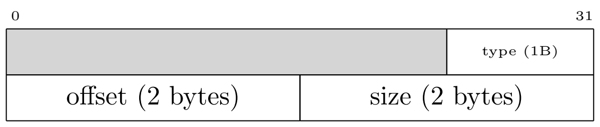

Figure 8: Data request payload (5 bytes).

3.4.6 Data reply (type `0x05`/`0x64`/`0x66`)

UDP data replies (type `0x05`) contain requested data. The data length is 1360 bytes, except if no more data is available. If a bot downloading a file receives a data reply with under 1360 data bytes, it assumes this is the last data block, and ends the download. If a data reply takes more than 5 seconds to arrive, the download is aborted. The maximum download size limit is 10MB, enforced by the receiving bot.

Each 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 use the same identifier. The file identifier is followed by the requested data. This structure is shown in Figure 9.

The transmitted files end with an RSA-2048 signature of the MD5 hash of the plaintext data, and are encrypted with an RC4 layer using a hardcoded key on top of an XOR encryption layer. Zeus only applies binary or configuration updates with version numbers strictly higher than its current version number. This means that it is not possible to make Zeus bots revert to older versions.

TCP data transfers start with a message header of type `0x64` for a binary update, or `0x66` for a configuration update, followed by the RC4 encrypted data. TCP data transfers are terminated by a message containing only a little-endian integer with the value 1 (no header).

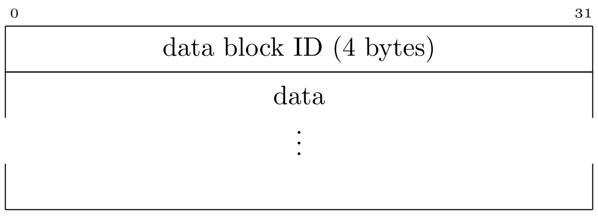

Figure 9: Data reply payload (length varies).

3.4.7 Proxy reply (type `0x06`)

Proxy replies return proxies in response to version requests with piggybacked proxy request markers. A proxy reply can contain up to 4 proxies, each of which is RSA-2048 signed.

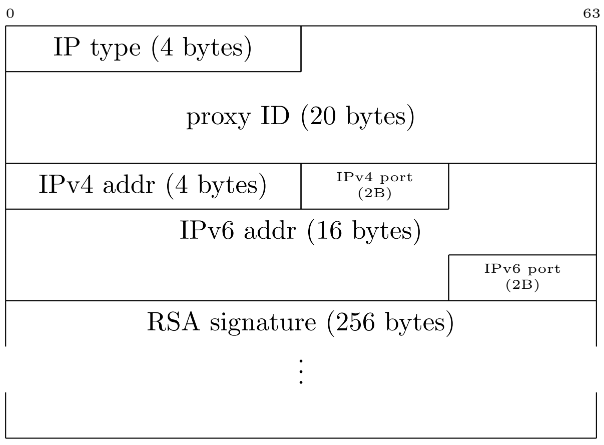

Figure 10: Proxy struct (304 bytes).

Each proxy entry is formatted as shown in Figure 10. The format is similar to that used in peer list replies, except that the IP type field is 4 bytes long instead of 1 byte, and there is an RSA signature at the end of each proxy entry.

3.4.8 Proxy announcement (type `0x32`)

Proxy announcements are similar to proxy replies, but are actively pushed through the Zeus network and are not sent in response to any message. When a bot is appointed as a proxy by the botmasters, it pushes a proxy announcement to all its neighbors to announce that it is now a proxy.

Proxy announcements utilize the TTL field in the Zeus header (see Section 3.3). The TTL field has an initial value of 4 for proxy announcements. In turn, each Zeus peer which receives a proxy announcement decrements the time to live value and forwards the announcement to each of its neighbors. This way, proxy announcements propagate very rapidly through the network, although they cannot reach NATed bots directly.

A proxy announcement contains a single proxy entry of the same format used in type `0x06` messages, as shown in Figure 10.

3.4.9 C2 message (type `0xCC`)

Unlike most message types, C2 messages are only exchanged between harvester bots and proxies, and are exchanged over TCP. C2 messages are used as wrappers for HTTP messages, and are encrypted with RC4 using the identifier of the receiving bot as the key. Because C2 messages wrap HTTP messages, we suspect that the communication between proxy bots and the C2 proxy layer is HTTP-based. The HTTP-based C2 protocol is virtually identical to the C2 protocol used in centralized Zeus [3, 5].

An example C2 HTTP header for a command request is shown in Figure 11. The X-ID field specifies the sub-botnet for which a command is requested.

```
POST /write HTTP/1.1
Host: default
Accept-Encoding:
Connection: close
Content-Length: 400
X-ID: 100
```

Figure 11: C2 HTTP header.

After the HTTP header comes an HTTP payload, which consists of several, optionally zlib-compressed, data fields. The payload begins with a header specifying the payload size and flags, and the number of data fields that follow. The payload header ends with an MD5 hash of the combined data fields, used to verify message integrity.

Each data field starts with a data header specifying the field type, flags, and compressed and uncompressed sizes. After the header comes the actual data, the structure of which is dependent on the type of data field. Each data field is encrypted with an XOR encryption.

C2 request messages typically contain several status and information fields about the requesting bot. Typical fields included in C2 requests are shown in Table 1. Note that the type numbers of data fields are completely independent from Zeus message type numbers.

type	content
`0x65`	system name and volume ID
`0x66`	bot identifier
`0x67`	infecting spam campaign
`0x6b`	system timing information
`0x77`	stolen data

Table 1: Typical C2 request fields.

The most important data field contained in a C2 response is the command field, which has type `0x01`. It contains an MD5 hash used to verify integrity of the command, followed by the command itself in the form of an ASCII-string. Several example command strings are listed in Table 2.

name	meaning
user_execute	execute file at URL
user_certs_get	steal crypto certificates
user_cookies_get	steal cookies
ddos_url	DDoS a given URL
user_homepage_set	set homepage to URL

Table 2: Example C2 command strings.

3.5 Communication Patterns

The previous sections have described the format and purpose of the Zeus message types. We now briefly clarify how these message types fit together and how Zeus bots typically behave. Zeus bots run a passive thread, which listens for incoming requests, as well as an active thread, which periodically generates requests to keep the bot up-to-date and well-connected. We describe the behavior of each of these threads in turn.

3.5.1 Passive thread

Every Zeus bot listens for incoming messages from other bots in its passive thread. A Zeus bot receiving an incoming request will handle this request to the best of its abilities, and attempt to send back an appropriate reply, as described in Section 3.4.

The sender of any successfully handled request is considered for addition to the receiving bot's peer list. This is the main mechanism used by *externally reachable* Zeus bots to learn about neighbors, and it is also how new bots introduce themselves to the network. If the receiving bot has fewer than 50 neighbors, then it always adds the sender of the request to its peer list. Additionally, if the identifier of the sender is already present in the peer list, then the corresponding IP address and port are updated. This is done to accommodate senders with dynamic IPs and discard stale dynamic IPs. If the identifier of the sender is not yet known, but the peer list already contains 50 peers or more, then the sending peer is stored in a queue of peers to be considered for addition during the next neighbor verification round (see Section 3.5.2).

Before adding a new peer to the peer list, a number of sanity checks are performed. First, only peers which have a source port in the expected range are accepted. NATed bots may make it into the peer lists of other bots, if they happen to choose a port in the valid range. Additionally, only one IP per /20 subnet may occur in a bot's peer list at once. If a potential new peer's IP is in a /20 subnet already represented in the peer list, it is not accepted. Recent versions of Zeus also include an automatic blacklisting mechanism, where each bot blacklists IPs that contact it too frequently in a specific (hardcoded) time window.

Most incoming messages are requests from other peers, and are handled by sending back the appropriate reply type. Type `0x32` proxy announcements are the exception. When a type `0x32` message arrives, its signature is checked for validity. If the message passes the check, the time to live field is decremented and the message is forwarded to all known neighbors if the time to live is still positive.

Furthermore, new proxies which pass verification are considered for addition to the receiving bot's proxy list. The proxy list is similar to the peer list, but is maintained separately. If the identifier of the new proxy is already in the proxy list, then the corresponding IP address and port are updated. Otherwise, if a proxy list entry over 100 minutes older than the new proxy is found, this entry is overwritten with the new proxy (this is not done for type `0x06` proxy replies). 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, effectively discarding the new proxy if the proxy list was already 10 entries long. It is worth noting that a Zeus bot will never add itself to its proxy list, even if it receives itself in a proxy announcement.

3.5.2 Active thread

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 itself, as well as the peer list and proxy list, up to date.

In each iteration of the loop, Zeus queries each of its neighbors for their binary and configuration file versions. This step serves to keep the bot up to date, and to check each neighbor for responsiveness. Also, if Zeus knows fewer than 4 proxies, it piggybacks a proxy request marker with each version request. Each peer is given 5 chances to respond to a version request. If a peer fails to answer within the maximum number of retries, Zeus checks if it has working Internet access by attempting to contact `www.google.com` or `www.bing.com`. If it does, the unresponsive peer is discarded. Else, Zeus stops attempting to probe the current peer. If the peer responded and is found to have an update available, the update is downloaded in a separate thread.

After version querying all peers in its peer list, Zeus proceeds to handle any pending peers which were queued from incoming requests (see Section 3.5.1). Pending peers are only handled if the peer list contains fewer than 50 peers, and is stopped as soon as the peer list reaches length 50. Each pending peer under consideration is sent a single version request, and is added to the peer list if it responds.

Finally, if the peer list contains fewer than 25 peers, the bot will actively send peer list requests to each of its neighbors until the peer list reaches a maximum size of 150 peers. This is only done once every 6 loop cycles (3 hours), and is an emergency measure to prevent the bot from becoming isolated. If, despite this effort, a bot does find itself isolated, it will attempt to recover by contacting its hardcoded bootstrap peer list. If this also fails, the bot will enter DGA mode, as further described in Section 4.

4 Domain Name Generation Algorithm

As mentioned in Section 3.1, Zeus contains a Domain Generation Algorithm (DGA), which is activated if a bot cannot reach any peers, or the bot cannot fetch updates for a week. Zeus uses the DGA-generated domains to download fresh RSA-2048 signed peer lists, and (in some variants) exchange C2 traffic. This section describes the Zeus Domain Generation Algorithm.

4.1 Algorithm Details

The Zeus Domain Generation Algorithm generates 1000 unique domain names per week. A bot wishing to use the DGA channel starts at a random position in the list of domain names for the current week and sequentially tries all domain names until it finds a domain which is responsive. Generated

```
 1 for(i = 0; i < 1000; i++) {
 2     /* S is a byte array and year, date, month are numeric */
 3     S[0] = (year + 48) % 256;  S[1] = month;
 4     S[2] = 7 * (day / 7);      S[3] = i;
 5     hash = md5(S);
 6 
 7     /* convert hash to URL domain name */
 8     name = "";
 9     for(j = 0; j < len(hash); j++) {
10         c1 = (hash[j] & 0x1F) + 'a';
11         c2 = (hash[j] / 8) + 'a';
12         if(c1 != c2) {
13             if(c1 <= 'z') name += c1;
14             if(c2 <= 'z') name += c2;
15         }
16     }
17 
18     /* select TLD for domain */
19     name += ".";
20     if(i % 6 == 0) {
21         name += "ru";
22     } else if(i % 5 != 0) {
23         if(i & 0x03 == 0) {
24             name += "info";
25         } else if(i % 3 != 0) {
26             if((i % 256) & 0x01 != 0) name += "com";
27             else name += "net";
28         } else {
29             name += "org";
30         }
31     } else {
32         name += "biz";
33     }
34 
35     domains[i] = name;
36 }
```

Figure 12: The P2P Zeus Domain Name Generation Algorithm.

domain names use top-level domains taken from the set {`biz`, `com`, `info`, `net`, `org`, `ru`}. The Zeus DGA bears some resemblance to the DGA used in Murofet (Murofet is a malware known to be related to centralized Zeus) [6].

The Zeus DGA is shown in C-like pseudocode in Figure 12. The code shown generates all 1000 domains for a given week. The generation of a domain name starts by taking the MD5 hash over the concatenation of (transformations of) the year, month, day, and domain index. As can be seen in lines 9 – 16, the MD5 hash is then used to generate a domain name which contains a variable number of characters between "a" and "z". Zeus domain names are thus variable-length strings containing at most 32 characters (excluding the top-level domain), all of which are lowercase letters. Finally, the domain is completed by selecting one of the six top-level domains and concatenating it to the domain name.

5 Conclusion

P2P Zeus is a significant evolution of earlier Zeus variants. Compared to traditional centralized versions of Zeus, P2P Zeus appears to be much more resilient against takedown and infiltration attempts. Potential countermeasures against P2P Zeus are complicated by its application of RSA-2048 to mission critical messages. The network's resilience is further increased by its use of a Domain Generation Algorithm backup channel, and (automatic) blacklisting and IP-restriction mechanisms. Furthermore, infiltration is complicated by the Zeus message encryption mechanism, which makes the use of randomized bot identifiers impossible in the general case.

6 Acknowledgements

We would like to thank Tillmann Werner and Christian Rossow for the collaboration on reversing the Zeus P2P protocol. We also thank Daniel Plohmann for his work on the Domain Generation Algorithm, Brett Stone-Gross for his work on the configuration file and C2 message formats, and Tomasz Bukowski and Christian J. Dietrich for their insights. This work was supported by the European Research Council Starting Grant "Rosetta" and the EU FP7-ICT-257007 SysSec project.

References

[1] Zeus Tracker. https://zeustracker.abuse.ch/.

[2] M. Antonakakis, R. Perdisci, Y. Nadji, N. Vasiloglou, S. Abu-Nimeh, W. Lee, and D. Dagon. From Throw-Away Traffic to Bots: Detecting the Rise of DGA-Based Malware. In *Proceedings of the 21st USENIX Security Symposium*, Bellevue, WA, USA, 2012.

[3] H. Binsalleeh, T. Ormerod, A. Boukhtouta, P. Sinha, A. Youssef, M. Debbabi, and L. Wang. On the Analysis of the Zeus Botnet Crimeware Toolkit. In *Proceedings of the 8th Annual Conference on Privacy, Security and Trust*, Ottawa, Ontario, Canada, August 2010.

[4] CERT.pl. Zeus P2P Monitoring and Analysis, 2013. Technical Report. http://www.cert.pl/PDF/2013-06-p2p-rap_en.pdf.

[5] N. Falliere and E. Chien. Zeus: King of the Bots, 2009. Technical Report, Symantec.

[6] K. Itabashi. How Trojan.Zbot.B!inf Uses the Crypto API, 2010. Technical Report, Symantec. http://www.symantec.com/connect/blogs/how-trojanzbotbinf-uses-crypto-api.

[7] A. Lelli. Zeusbot/Spyeye P2P Updated, Fortifying the Botnet, February 2012. Technical Report, Symantec. http://www.symantec.com/connect/blogs/zeusbotspyeye-p2p-updated-fortifying-botnet.

[8] 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*, 2002.

[9] C. Rossow, D. Andriesse, T. Werner, B. Stone-Gross, D. Plohmann, C. Dietrich, and H. Bos. P2PWNED: Modeling and Evaluating the Resilience of Peer-to-Peer Botnets. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, May 2013.

[10] B. Stone-Gross. The Lifecycle of Peer-to-Peer (GameOver) Zeus, July 2012. Technical Report, Dell SecureWorks. http://www.secureworks.com/cyber-threat-intelligence/threats/The_Lifecycle_of_Peer_to_Peer_Gameover_ZeuS/.

[11] abuse.ch. Zeus Gets More Sophisticated Using P2P Techniques, 2011. Technical Report. http://www.abuse.ch/?p=3499.

[12] J. Wyke. What is Zeus?, 2011. Technical Report, Sophos.