PEERING INTO SPAM BOTNETS

Jarosław Jedynak & Maciej Kotowicz
CERT Poland

Email {jedynak.jaroslaw, kotowicz.maciej}@gmail.com

ABSTRACT

Someone recently said that the exploit kit landscape is dying – but this certainly is not the case for the spamming industry, which, over the last year, was probably the biggest source of infections worldwide.

Despite spam botnets being so important in the lifecycle of malware, recent publications describing massive spam operations (which can be counted on the fingers of one hand) have either skipped over the technical details or else concentrated too much on high-level aspects for our liking.

In this paper, we will describe a few of the most prominent spam botnets out there: Emotet, Kelihos, Necurs, Send-Safe and Tofsee.

Our main goal is to describe the technical details of the network protocols used by these botnets to communicate with peers and C&Cs. We will show how to dissect and reimplement basic communication (including, but not limited to, receiving new spam, downloading malicious attachments and detecting updates to the core spam bot or its sub modules).

We will also present the results of our monitoring, including how and what is being spammed, and by which botnet. We will conclude with some funny quirks that one can find while looking closely at spamming operations.

Since these botnets aren't new, there is some information about them already in circulation. Unfortunately this knowledge is often fragmented, or hidden within companies. With our talk we hope to share our insights, and allow everyone to track and destroy spam on their own.

1. INTRODUCTION

Spam is probably the biggest vector of infection for both commodity and targeted malware. One of the reasons it has earned this position is thanks to spam botnets, malware that has only one job: to send as many malicious mails as possible. Most of these beasts operate for years and are very resilient to takedowns, largely due to their complicated infrastructure and protocols. It's possible to find analyses of the protocols used by some more notorious spammers, but most of them are quite outdated. In this paper we aim to provide a detailed description of how these malware communicate and how we can leverage this information to gain a better understanding of their operations and get more malware directly from the source.

We hand picked a few families that either have been engaged with spamming for very long time or whose protocol is unique and outstanding. The list comprises:

- Emotet[1]
- Kelihos
- Necurs
- Send-Safe
- Tofsee

[1] Its spamming module to be precise, and its newest version.

2. EMOTET

Emotet is an offspring [1] of the long-lived malware family that started with Cridex and allegedly give rise to such malware as Dridex and Dyre. It appeared in June 2014 [2], targeting clients of German banks. It was, and as far we know still is, distributed only by spam emails that originate from previously infected machines. In the past it was trivial to distinguish Emotet's malspam from others – the emails always impersonated DHL shipment orders and had very unique URL patterns under which the malware was located.

Example URL [3]	**Purpose**
http://freylau.de/VMfWxYqJme	First landing page
www.buziaki.gorzow.pl/CgHsoRfvpqGk2/8114721795964851.zip	Redirect
Dhl_Status_zu_Sendung_340839262847_____ID_S01_DHL__DEM06_MS02_06_2015___A23_21_15.exe	Final malware

Table 1: URL patterns found in Emotet malspam.

Today, they have shifted their tactics and are using a more generic approach, dropping *Word* documents that contain a PowerShell command responsible for downloading and executing Emotet.

While we didn't analyse closely how the spamming module operated in the past, based on how general C&C communication changes, we can assume that it had very little in common with today's protocol. During our research we found Emotet's protocol, while rather simple, quite fascinating. We won't delve into the details of it here, since we have already described it on our blog [4], but recall that it is an AES-encrypted blob of binary data.

Based on educated guesses, we discovered that the binary blob appearing in communication was, in fact, a modified version of *Google*'s proto buffers. At the time of writing we are not sure if the modification came from sloppy custom implementation or by other means. For the purpose of our analysis we assumed that it was a deliberate move.

2.1 Spam operation

While most spamming malware is designed to behave like an SMTP client – the malware communicates directly with the email servers of its victims – the authors of Emotet took a different approach. Many properly configured SMTP servers either blacklist or greylist messages from untrusted or unknown sources – these mechanisms were introduced to prevent users from receiving the bulk of unsolicited messages.

To work around that, Emotet uses trusted services like *Gmail*, *Yahoo* and *Live.com* as its relays, abusing firstly stolen credentials, for which there is a separate module. This is clearly visible in configuration data received from the C&C:

```
id: 2075010
mail_server: "sd-smartermail.sdsolutions.de"
port: 25
```

```
login: "info@prodia-bamberg.de"
password: "xxxxxxxxx"
email: "info@prodia-bamberg.de", id: 2129057
mail_server: "mx1.kabsi.at"
port: 587
login: "h2199a00"
password: "xxxxxx"
email: "h2199a00@kabsi.at", id: 2136311
2Dhl_Status_zu_Sendung_340839262847_____ID_S01_DHL__
DEM06_MS02_06_2015___A23_21_15.exe
mail_server: "host72.kei.pl"
port: 587
login: "dyspozytor-pdz@kolprem.pl"
password: "xxxxxxxxxx"
email: "dyspozytor-pdz@kolprem.pl",
```

3. KELIHOS

Kelihos, also known as Hlux, is one of the older spam botnets. It was first discovered around December 2010. Finally, in April 2017, after many previous attempts to take it down, the botnet operator was arrested, and the FBI began sinkholing the botnet [5]. For this reason, this part of the paper is provided mostly for historical reasons – the techniques described probably won't work, because the peers are dead (unless Kelihos comes back from the dead in the future). Nevertheless, we think that this botnet is interesting enough that it still deserves a mention here. A great write-up on Kelihos communication can be found on the *Fortinet* blog [6]. The scope of this paper is very similar, though we focus more on the implementation side, and provide a few bits of code. We also think that Kelihos's unusual approach to encryption is interesting to read about.

3.1 Peer handshake

Kelihos uses surprisingly solid cryptography in its P2P communication – each bot has its own key pair (generated using the Crypto++ library). Communication is encrypted using asymmetric cryptography, and because of this, it's impossible to decrypt it, even when the whole traffic is captured.

When Kelihos wants to perform a key exchange with a peer, it generates a 16-byte random key, and signs it with its private key with PKCS1 RSA/SHA1.

The handshake message contains this random data, a signature for it, and a public key. Kelihos packs these fields using a simple structure, presented in Figure 1.

The handshake can be generated with the help of the following Python code:

```
flags = 'e78673536c555545'.decode('hex') # timestamp
and magic
blocks = '03104840'.decode('hex')
# [0x03, 0x10, 0x48, 0x40]
# 3 blocks, 16 bytes of random data, 0x48 bytes of
public key, 0x40 bytes of signed data
hdr = flags + blocks

randbytes = Random.new().read(16)
pubkey = rsakey.publickey().exportKey(format='DER')
[20:]
hash = hashlib.sha1(randbytes).digest()
pad = '01ffffffffffffffffffffffffffffffffffffffffff
ffffffff00'.decode('hex')
```

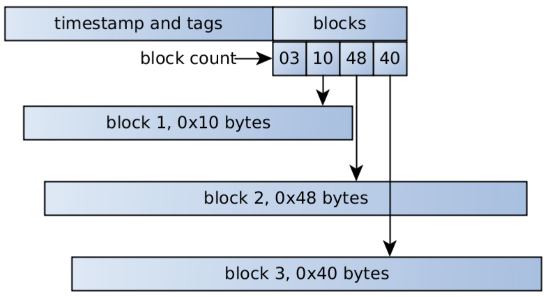

Figure 1: Peer handshake packet structure.

```
sha1_designator = '3021300906052b0e03021a05000414'.
decode('hex')
signature = rsakey.sign(pad + sha1_designator +
hash, '')[0]
signature = number.long_to_bytes(signed)
handshake_request = hdr + randbytes + pubkey +
signature
```

Receiving data is more complicated – data is first encrypted using a Blowfish cipher in CBC mode, and after that, we have a similar structure (three blocks, with random data, public key, and signature). Example decryption code:

```
data = sock.recv(100000)

rsa_enc, blowfish_enc = kelihos_get_blocks(data) #
parse blocks - response has two ones
blowtmp = rsakey.decrypt(rsa_enc)
blowkey = blowtmp[-16:]

print 'KELIHOS_HANDSHAKE_RESPONSE'
print ' - rsa_encoded', rsa_enc.encode('hex')
print ' - rsa_decrypt', blowtmp.encode('hex')
print ' - rsa_result', blowkey.encode('hex')
print ' - blowfish_enc', blowfish_enc.encode('hex')

iv = blowfish_enc[1:9]
cipher = Blowfish.new(blowkey, Blowfish.MODE_CBC, iv)
msg = cipher.decrypt(blowfish_enc[9:])

some_data = msg[:12]
rsa_key = msg[12:12+72]
signeddata = msg[12+72:12+72+64]

print ' - blowfish_dec', msg.encode('hex')
print ' - some_data', some_data.encode('hex')
print ' - rsa_key', rsa_key.encode('hex')
print ' - signeddata', signeddata.encode('hex')

key = RSA.importKey(rsa_key)
sign = key.encrypt(signeddata, 'ignored')[0]

print ' - sign', sign.encode('hex')
print ' - hash', sign[-20:].encode('hex')
```

This mechanism of key exchange is a good example of correctly used asymmetric crypto – it actually made the analysis of traffic harder, because we needed to dump private keys if we wanted to analyse raw traffic.

3.2 Peer list exchange

During peer data exchange, Kelihos serializes all relevant information into a certain big structure, which is then encrypted.

In contrast to the handshake, most encryption methods used here are home-made by the malware's authors and are cryptographically not very sound.

Most interesting, and a quite unique idea, is the use of eight encryption algorithms in a random order determined by a random 16 bytes from the header.

First, a list of encryption functions is created and shuffled (with a random generator seeded by a 16-byte header). The seeding algorithm looks like this:

```
void crypto_function_list(crypto_function_list
*crypt, string_t *seed)
{
  strc = init_temporary_structure();
  strc.str = seed;
  strc.offset = 0;

  list_insert_by_string_offset(&strc, func_xor_1byte);
  list_insert_by_string_offset(&strc, func_viscrypt);
  list_insert_by_string_offset(&strc, func_mathops);
  list_insert_by_string_offset(&strc, func_bitcrypt1);
  list_insert_by_string_offset(&strc, func_pairwise_
  swap);
  list_insert_by_string_offset(&strc, func_simple);
  list_insert_by_string_offset(&strc, func_reverse);
  list_insert_by_string_offset(&strc, func_bitcrypt2);

  it = strc.list.tail->prev; // cyclic list
  while (it != strc.list.tail)
  {
    append_list(crypt, &it->data);
    it = it->prev;
  }
  free_list(&strc.list);
  return crypt;
}
```

Where `list_insert_by_string_offset` is using consecutive characters from the seed as offsets for inserting functions in random order into the function list.

```
void list_insert_by_string_offset(temporary_struct
*strc, crypto_function func)
{
  seed = strc->str->length ? *getCharAtPos(strc->str,
  strc->offset) : strc->list.size;
  offset = seed % (strc->list.size + 1);
  list_insert_at_posiion(&strc->list, offset, func);
  if (++strc->offset >= strc->str->length) strc->offset
  = 0;
}
```

After that, functions are called consecutively in plaintext.

Name	Description
`xor_1byte`	XOR every byte in the string with the same byte
`viscrypt`	Visual crypt algorithm (XOR `string` with `string[1:]+chr(len(string))`)
`mathops`	Meaningless mathematical operations on every byte (see Appendix)
`bitcrypt1`	Meaningless bitwise operations on every byte (see Appendix)
`bitcrypt2`	Meaningless bitwise operations on every byte (see Appendix)
`pairwise_swap`	swap(string[0], string[1]), swap(string[2], string[3]), swap(string[4], string[5]), . . .
`simple swap`	Nibbles in every byte
`reverse`	Reverse string

Table 2: Functions.

Non-obvious encryption methods are shown in Appendix A.

All these encryption functions are trivially decryptable with a bit of cryptanalysis. It's possible that the malware creators think that combining multiple weak encryption algorithms will create a strong one, but we believe that this is just an attempt at obfuscation and slowing researchers down, not really a proper encryption scheme. Especially since, after that, standard Blowfish encryption is used again (with a random 0x10 bytes as a key). Finally, the Blowfish key is encrypted with the remote peer's public key.

Now the malware creates three data blocks:

- random bytes determining the decryption function order
- encrypted Blowfish key
- encrypted peer list

The first block is additionally encrypted with the `viscrypt` and `bitcrypt1` methods, then a few bytes of random data are prepended to it, and finally, one byte with obfuscated length of that random data is prepended.

All three blocks are concatenated, and encrypted with the `bitcrypt1` method, just in case.

After that, the length of every block is packed into the header. The header contains six DWORDs, with the following meanings:

- block 1 length = HEADER[1] - HEADER[0]

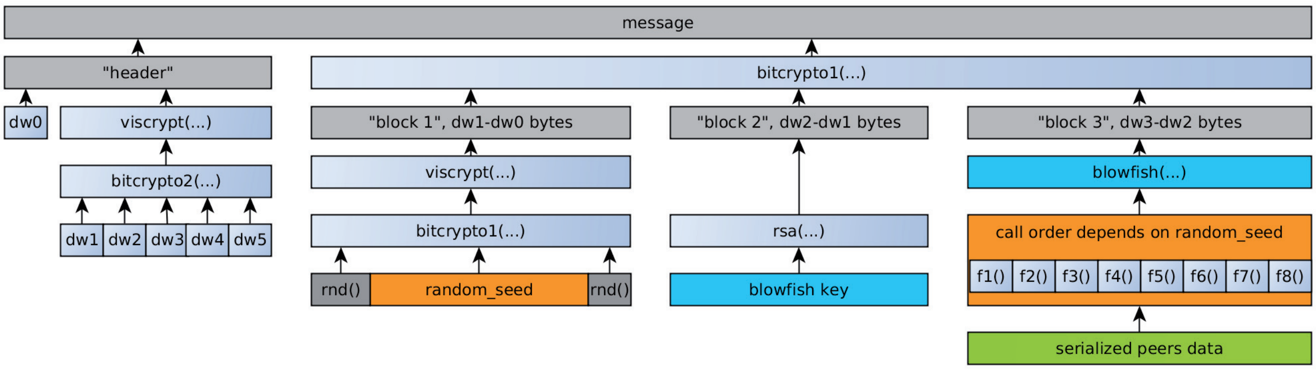

Figure 2: Kelihos encryption method.

- block 2 length = HEADER[2] - HEADER[1]
- block 3 length = HEADER[3] - HEADER[2]
- unk1 length = HEADER[4] - HEADER[3] - 95
- message type = HEADER[5] - HEADER[4] - 197

All but the first four bytes of the header are additionally encrypted with the `viscrypt` and `bitcrypt2` methods. This probably sounds really convoluted and complicated – because it is. While using asymmetric cryptography and Blowfish is a good idea, we don't see any reason for all the other complicated steps – unless the malware creators just wanted to waste researchers' time. The whole encryption process is summarized in Figure 2.

If we want to decrypt data we need to go through all these steps, but in reverse. First, we have to decrypt the 'header', and compute the block lengths. After that, we decrypt all three blocks using `bitcrypt1`, and recover the Blowfish key and random seed. Finally, we decrypt serialized peers data using that key and seed. A commented routine for most of this operation can by found in Appendix A.

4. NECURS

Necurs is one of the biggest botnets in the world – with more than 1.5 million infected computers, it has active bots in almost all countries, several hundred thousand of which are online at any given time.

Compromised machines usually send spam emails to a large number of recipients, though the botnet has the capability to act as a proxy or perform DDoS attacks.

4.1 High-level overview

Necurs' communication protocol is complicated, definitely not pretty, and full of strange quirks [7]. For example, three different compression algorithms are used, encryption algorithms are home-made and serve more for obfuscation than securing transmission, and a lot of structures are unnecessarily bloated.

The Necurs botnet is divided into sub-botnets – each Necurs binary has a hard-coded and immutable `botnet_id` saved in its resources. Sub-botnets have different C&C servers, peer lists, and DGA (`botnet_id` is used as part of DGA seed).

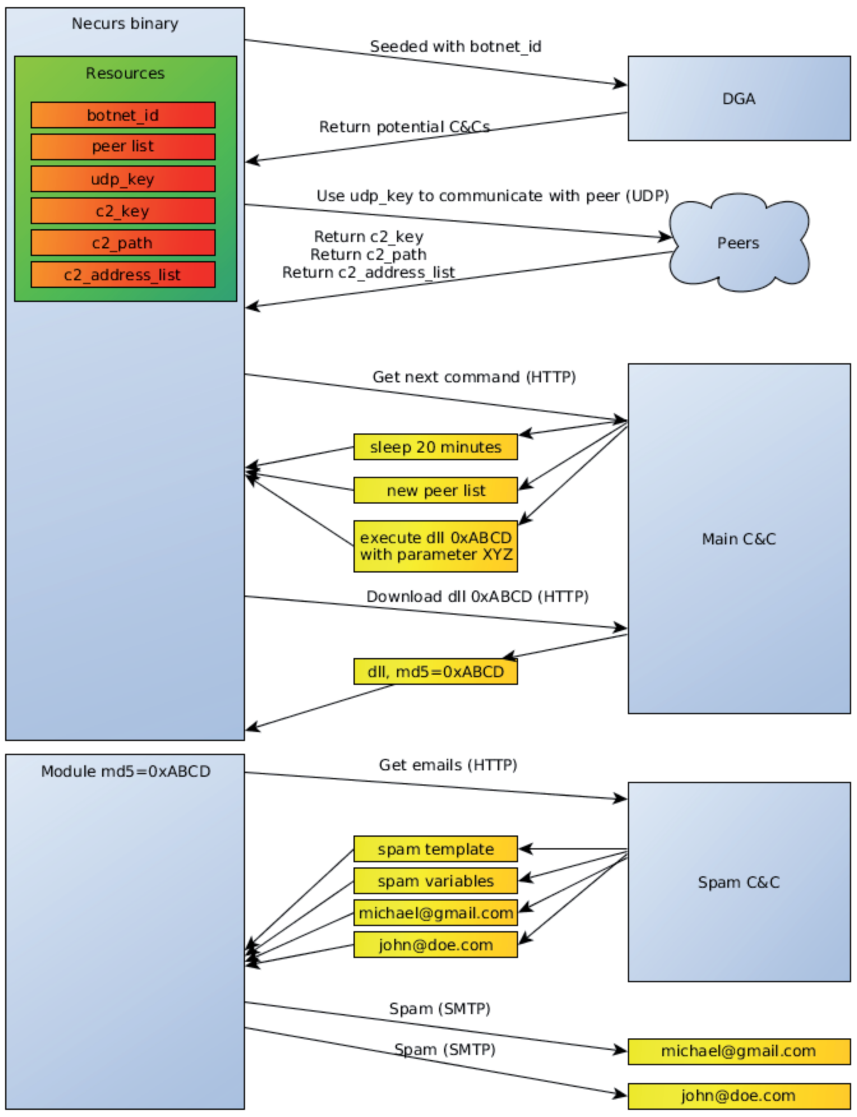

Figure 3: Necurs communication.

Currently, we know of three botnets: ID `5` (the biggest), `9` (a smaller one) and `7` (small, and C&C is long since dead).

The botnet is an example of a hybrid network, i.e. a mixture of centralized (that simplifies and speeds up management) and peer-to-peer decentralized models (making it much more resistant to takedowns) – and additionally, DGA is implemented. With so many features it's no wonder that Necurs had survived so long.

The malware attempts to connect to the C&C server, whose IP address is retrieved in a number of different ways:

- First, a couple of domains or raw IP addresses are embedded in the program resources.
- If the connection fails, Necurs runs a domain generation algorithm, crafting up to 2,048 pseudorandom names, the generation of which depends on the current date and seed hard coded in encrypted resources, and tries them all in a couple of threads. If any of them resolves and responds using the correct protocol, it is saved as a server address.
- If all these methods fail, the C&C domain is retrieved from the P2P network – the initial list of about 2,000 peers (in the form of IP+port pairs) is hard coded in the binary.

During our analysis, Necurs used the last method, since none of the DGA domains were responding. It is, however, possible that in the future the botnet's author will start to register these domains – a new list of potential addresses is generated every four days.

After establishing a successful connection to the C&C, Necurs downloads (using a custom protocol over HTTP) the required information, most notably additional modules (spam module, proxy module, rootkit) and additional C&Cs (for example the spam C&C). After that, each module is started. Finally, the spam module requests templates and variables from the spam C&C. When all the necessary information is downloaded, spam is sent. This process is summarized in Figure 3.

4.2 Binary resources

If we want to start communicating with Necurs, we first have to decrypt its resources. They are stored in binary in encrypted form. To find them, we have to find two consecutive qwords in memory that satisfy the following equation:

```
a * 0x48F1398FECF + 12345678901253 === b (mod 2**64)
```

They mark the first bytes of encrypted resources. In Python it's a simple five-liner:

```
def get_base(dump):
 for i in range(len(dump) - 0x10):
   a, b = struct.unpack("<QQ", dump[i:i + 0x10])
   if (a * 0x48F1398FECF + 12345678901253) &
   0xFFFFFFFFFFFFFFFF == b:
     return i
```

After that, we have the `resourceList` structure in memory:

```
struct resourceList {
  uint64_t marker_qword_1;
  uint64_t marker_qword_2;
  uint8_t encrypted_sizes[8]; // uint32_t compr_size,
  raw_size;
  singleResource resources[N]; // array of resources
};

struct singleResource {
  uint32_t size_times_256;
  uint64_t id;
  uint8_t data[]; // "size" bytes
}
```

Where `id` is `resource id`, see Table 3. But the `resources` array and `encrypted_sizes` are encrypted in memory (and potentially compressed with the APLIB32 algorithm) – we have to decrypt them first:

```
def next_key(k):
 k *= 0x19661f
 k += 0x3c6ef387
 k &= 0xFFFFFFFF
 return k

def decrypt_resources(dump):
 base = 0
 key = struct.unpack("<I", dump[base:base + 4])[0]

 encrypted_sizes = dump[base+0x10:base+0x18];
 for i in range(8):
   encrypted_sizes[i] ^= next_key(key) & 0xFF
 for i in range(base + 0x18, len(dump)):
   key = next_key(key)
   dump[i] = chr(ord(dump[i]) ^ key & 0xFF)

 compressed_size, real_size = struct.unpack('<II',
 encrypted_sizes)
 if compressed_size != real_size:
   dump = aplib.decompress(dump[0x18:])
```

After that, we should have decrypted resources in memory, and parsing them is almost trivial:

```
base = 0x18 # skip three first QWORDs
resources = []
while True:
   sz, id = struct.unpack("<IQ", dump[base:base +
   12])[0]
   sz >>= 8

   if not id:
     break

   res = dump[base + 12:base + 12 + sz]
   resources.append({"offset": base, "id": id,
   "content": res})
   base += 12 + sz
```

The most interesting resource types are presented in Table 3.

Id	Meaning	Example
`0x5148b92048028c4e`	Botnet ID	5
`0x59e80beb0279afba`	Peer list	. . .
`0x2f26c75348f3f531`	P2P communication key	. . .
`0x7c7b239242b0aec2`	C&C communication key	. . .
`0x6fa46c4146c2c285`	C&C URL path	/forum/db.php
`0x7ddd7ae7c4e9d441`	C&C domain	npkxghmoru.biz

Table 3: Necurs resources.

4.3 DGA and P2P

Now we need the C&C server address. As we noted, there are three ways to get it. If it's stored in static resources, we already have it. Unfortunately, this is often not the case (or the one stored is obsolete) and we need to resort to other techniques.

The second option is a DGA algorithm. The domain list changes every four days, and depends only on the current date and botnet ID:

```
def dga_mix_and_hash(param):
 param %= 2 ** 64
 for i in range((param & 0x7F) + 21):
    param = (param + ((7 * param) ^ (param << 15)) +
    8 * i - (param >> 5)) % 2**64
 return param

def dga_generate_one(year, month, day, seed, rnd_
param):
domain = ""
 mix = dga_mix_and_hash(year)
 mix = dga_mix_and_hash(month + mix + 0xAAAA)
 mix = dga_mix_and_hash((day >> 2) + mix)
 mix = dga_mix_and_hash(rnd_param + mix)
 nxt = dga_mix_and_hash(seed + mix)
 for i in range((mix % 15) + 7):
   nxt = dga_mix_and_hash((nxt >> 32 << 32) + (nxt %
   2**32) + i)
   domain += chr(ord('a') + nxt % 25)
   nxt = dga_mix_and_hash((nxt >> 32 << 32) + (nxt %
   2**32) + 0xABBEDF)

 tld_ndx = ((nxt >> 32 << 32) + (nxt % 2 ** 32)) % 43
 return domain + "." + [
   "tj", "in", "jp", "tw", "ac", "cm", "la", "mn",
   "so", "sh", "sc", "nu", "nf", "mu", "cx", "cc",
   "tv", "bz", "me", "eu", "de", "ru", "co", "su",
   "pw", "kz", "sx", "us", "com", "net", "org",
   "biz", "xxx", "pro"
 ][tld_ndx]
```

In practice, malware creators have never used this technique (as far as we know), and it's used solely by malware researchers for tracking purposes.

Finally, the most reliable and most useful method for getting the C&C address is to ask the P2P network for it. All P2P communication happens over UDP protocol. The outermost layer of communication looks like this (as C-structure):

```
struct outer_layer{
  uint32_t key;
  uint32_t checksum;
  uint8_t data[];
};
```

This data is encrypted using a key calculated as a sum of the key field and the first 32 bits of the public key contained in file resources. This home-made encryption algorithm is equivalent to the following Python code:

```
def rolling_xor(outer_layer):
 msg = outer_layer.data
 check = outer_layer.key
 buff = ""
 for c in msg:
    c = chr(ord(c) ^ check & 0xff)
    buff += c
    check = (check + (8 * (check + 4 * ord(c))) ^
    ror4(check, 7)) % 2**32
 assert outer_layer.checksum == check
 return buff
```

Inside the outer layer we have real messages, wrapped into another small structure:

```
struct inner_layer {
  uint32_t size_flags; // packed size and flags (length
  << 4 | flags)
  uint8_t data[];
};
```

The most interesting message type is a greeting/handshake:

```
struct greeting{
  uint32_t time; // Milliseconds since 1900-01-01
  uint64_t last_received; // ID of last received
  message - zeroes intially
  uint8_t flags;
};
```

And the response should look like this:

```
struct response{
  uint32_t version_low;
  uint8_t version_high;
  uint8_t size[3]; // Little Endian
  resourceList resources;
  uint8_t signature[];
};
```

The whole message is signed using a key from file resources. The most important part of this structure is the resource list, which is in the same format as those stored inside executables. Interestingly, peers don't send new neighbourhood lists – these are sent by the C&C itself. The most likely reason for this measure is to avoid P2P poisoning since it is known that the peer list received from the main server is authorized and correct.

4.4 C&C communication

The C&C protocol is vaguely similar to the P2P one, but the encryption routines and structures it uses are a bit different – also, the underlying protocol is HTTP (POST payload) instead of raw UDP sockets. The first stage is exactly the same (outer_layer structure), with different constants in the encryption algorithm:

```
def xor_encrypt(outer_layer):
  res = outer_layer.key
  buf=""
  for c in outer_layer.data:
     c = ord(c) ^ res & 0xff
     res = (res + (2 * (res + 4 * c)) ^ ror4(res, 13))
     % 2**32
     buf += chr(c)
  assert res == outer_layer.checksum
  return buf
```

But after decryption we get another structure:

```
struct cc_structure{
  uint64_t random_data; // random 8 bytes, probably
  to increase entropy
  uint64_t botID;
  uint64_t millis_since_1900;
  uint8_t command; // 0 - get command, 1 - download
  file, 2 - ping.
  uint8_t flags; // 1 - RSA sign, 2 - compress, 4 -
  timePrecision
  uint8_t payload[];
};
```

The contents of the payload field (perhaps compressed, depending on the second bit of flags) depends on the message type (command field):

- If command == 1 (download file), the payload is simply an SHA-1 hash of the requested file.
- If command == 0 (get command request), the payload structure is much more complex – again, a list of resources, but with a different structure.

Every resource has the following header:

```
struct cc_resource{
  uint8_t type;
  uint64_t id;
  uint8_t data[];
};
```

Where id is request/response id. Tables 4 and 5 contain possible requests and responses that the bot can send and receive.

Id	Meaning
0x4768130ffd8b1660	Botnet Id
0x50a29bce1ea74ddc	Seconds since start
0x5774f028d11237ac	System language
0xc3759a8411bcfb90	Public IP
0xd8cc549b8fb48978	Is user admin?
0x0a8aa0eec8402790	Is win64?
0xa6f73a722b8d2144	Is rootkit installed?
0x9924541302c75f90	Public TCP port for P2P
0x543591d7e21cfc94	Current hash of peer list

Table 4: IDs used in HTTP request commands.

Id	Meaning
0x4008cdaf91d42640	P2P Peer list
0x49340b1574c451a4	HTTP C&C domain list
0xd2b3cb6d2757a62c	Sleep for N seconds
0xf7485554ea9dfc44	Download and execute module
0x3cae696275cd12c4	Download and execute rootkit

Table 5: IDs used in HTTP response commands.

Data depends on resource type:

```
struct cc_resource_type_0 {
  uint32_t size;
  uint8_t data[]; // length=size
};

struct cc_resource_type_1 {
  uint32_t data;
};

struct cc_resource_type_2 {
  uint64_t data;
};

struct cc_resource_type_3 {
  uint64_t data;
};

struct cc_resource_type_4 {
  uint16_t size;
  uint8_t data[]; // length=size+1
};

struct cc_resource_type_5 {
  uint8_t data[20];
};
```

Type 4 is usually used to send text data, which is probably the reason why the resource size is increased by one (for null terminator). A client sends a list of such resources to the C&C. We were able to identify the meaning of some of them:

- DGA seed
- Number of seconds since malware start
- Unix timestamp of malware start
- OS version and its default language
- Computer's IP (local if behind NAT)
- UDP port used to listen for P2P connections
- Custom hash of current peer list.

The server responds with a very similar format, depending on command type:

- If command == 1, response is just the requested file contents (usually compressed, depending on flags).
- If command == 0, response is again more complicated – a list of resources in the same format as in the request.

One of the more interesting resources that we can receive from the server is a new peer list (if we sent a hash that doesn't match one in the C&C) or a new DLL announcement.

The latter resource again has its own structure for communication purposes, also made up of concatenated sub-resources of the following form:

```
struct subresource{
  uint32_t size;
  uint8_t unknown[18];
  uint8_t sha1[20];
  char cmdline[]; // length=size-42
};
```

The command should be interpreted as a request for running a DLL identified by its SHA-1 with command line parameters stated in the cmdline field – in practice, the argument is a newline-separated list of C&C addresses (with HTTP path) to be connected to.

4.5 Spam module – communication

The last protocol we will describe (but a very important one), is the communication of the downloaded DLL module, whose responsibility is to send spam emails.

The information is wrapped in the following structure (sent as POST data over HTTP):

```
struct spam_wrap{
  uint8_t data[];
  uint32_t crc32;
  uint32_t key; // 4th bit of key is compression flag.
};
```

The encryption algorithm used is as follows:

```
def encrypt(msg, key):
  key=rol4(key, 0x11)
  res=""
  for c in msg:
   tmp=ror4(key, 0xB)
   key+=((0x359038a9*key)^tmp)&0xFFFFffff
   res+=chr( (ord(c)+key) & 0xFF )
return res
```

So messages are packed like this:

```
def send_message(json):
   rnd = rtdsc()
   enc = encrypt(json, rnd)
   checksum = rol4(crc32(enc) ^ rnd, 5)
   payload = enc + struct.pack("<I", checksum) +
   struct.pack("<I", rnd)
   return requests.post(dom, data=payload,
   timeout=30)
```

An exemplary JSON for spam request can look like this:

```
js = {
   "vmjSIoC": guid,
   "WoVEf3A": "zOPeFRx",
   "GDncpsW": {
     "gzAfKVf": True,
     "Qet4BWy": "my_domain.tld",
     "6G18OEO": 0,
     "tGeZADS": []
   },
   "dg3XGB9": current_unix_timestamp
}
```

After decryption, we receive raw data as a JSON string (unless the compression flag was set, in which case the data needs to be unpacked – as we found out, a QuickLZ library was used in the malware for this purpose). The following is a sample JSON string:

```
{
  "vmjSIoC": -3740355867459618972,
  "nCZ1DIN": {
     "3ud2qDx": ["k***@jacob*****.com",
     "pranav*******@yahoo.com", (...) ],
     "kLhlsvR": "%%var boundary =
     b1_{{lowercase(rndhex(32,32))}}(...)",
     "5U6ci2Y": {
        "body": {
           "R9Y2jrb": 3730515652,
           "Ew7Rtuh": 339
        },
        (...)
        "wikibook.003": {
           "R9Y2jrb": 2392427997,
           "Ew7Rtuh": 99328000
        }
     },
     "LDB53Ml": false,
     "4aukyxg": 50,
     "9LVmdDs": 1,
     "6G18OEO": 677299251,
     "dcatsQu": 3,
     "5xTnygD": 8,
     "Wmto8rv": 21600,
     "jdTJLPh": 3,
     "LsHwjQC": 600,
     "lm74D93": 86400
  }
}
```

Unfortunately, keys are obfuscated, so we had to guess their meaning.

Id	Meaning
3ud2qDx	Spam target addresses
kLhlsvR	Spam templates
5U6ci2Y	Spam resources (variables)

Table 6: Meaning of the keys.

Finally, one of the fields in the received dictionary contains a script used to generate randomized emails (like on the top of the post), and as another field – a list of parameters passed to this script (e.g. eng_Names). We can make a separate request to download the values of these arguments – as a response, we will receive, for example, a list of English names to be substituted, or a few base64-encoded files to be used as an attachment.

4.6 Proxy/DDoS module – communication

There is another functionality hidden in Necurs – not used as often as the spam module, but still present.

It was described in great detail on the *Anubis Networks* blog [8], so we'll just go over the most important things.

The first thing the proxy module does is to check if it's behind a NAT. This is done by querying the external API (checkip.dyndns.org or ipv5.icanhazip.com) and comparing it with the local IP address.

After that, the bot measures the available bandwidth (by downloading *Windows 7* SP1 from *Microsoft* and measuring the time taken), and computes the `bot_id` (using the same algorithm as the main module).

If the system is not behind a NAT, the proxy module starts a SOCKS/HTTP proxy service listening on a random port.

After that, the module starts connecting to the C&C server in a loop and sends a beacon every 10 seconds. The C&C server can respond with a few different commands:

- Type 1: Computers usually are behind a NAT, so additionally a 'connectback proxy' is implemented. After this message, the connection socket is reused, so the proxy can work even behind a firewall.
- Type 2: Sleep (the bot will sleep for five minutes)
- Type 5: DDoS – the bot will start a DDoS attack against a specified target.

Implemented attack types are HTTP flood and UDP flood.

4.7 Tracking

We tried to start tracking Necurs in early winter 2017, but we had a lot of problems with bootstrapping our trackers because of the period of inactivity that Necurs was going through. We only managed to start at the beginning of February 2017 – the botnet was increasingly active from then until now. Captured changes are presented in Figure 4.

According to our data, the big changes in the C&C infrastructure correspond more or less to bigger waves of spam activity.

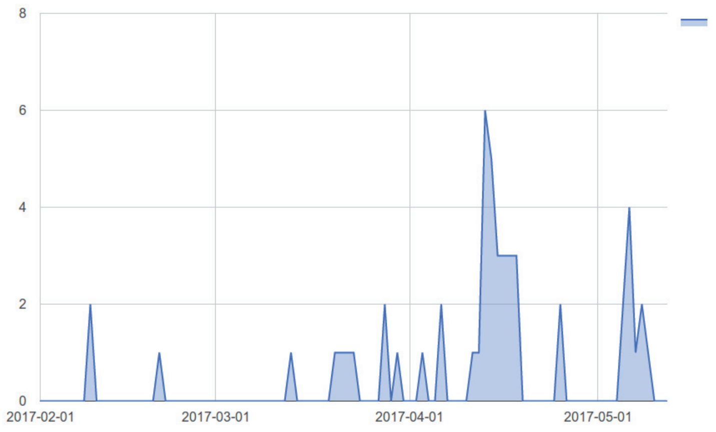

Figure 4: Number of changes in C2 configuration per day.

5. SEND-SAFE

Send-Safe is a notorious spamming tool, nowadays used mostly by the man1 [9] group. The history of Send-Safe goes by back to 2002, to a domain, send-safe[.]com, and operations run by Ruslan Ibragimov [10], but we believe it was rewritten, probably based on leaked code, and weaponized to be a spam bot rather than a spam tool. Searching through *VirusTotal* we found the first sample [11] of this strain uploaded around March 2016, and *Dr.Web* [12] started to detect it as Trojan.Ssebot.1 on 5 April 2016.

```
Date:   0x56F4D6A5 [Fri Mar 25 06:11:49 2016 UTC]

LegalCopyright: (c) 2005-07 Send-Safe
InternalName: Send-Safe Enterprise Mailer
FileVersion: 2,5,0,854
CompanyName: Send-Safe
ProductName: Send-Safe Enterprise Mailer
ProductVersion: 2,5,0,854
FileDescription: Send-Safe Enterprise Mailer
OriginalFilename: sse.exe
```

5.1 Communication

Tracking Send-Safe operations is not an easy task, mostly due to the design of its C&C protocol. Its authors decided that the best way to remain stealthy was to keep the main channel closed for most of the time, and open it only when they were ready to send spam. This concept is achieved by splitting the C&C communication into two parts:

- short UDP messages to inform the operators that the malware is alive
- normal HTTPS requests to receive information about spam targets and content of messages.

To make things a little bit simpler, both services are hosted on the same IP address, only the ports are different.

5.1.1 Configuration

Before we go into the details of the communication protocols, here is a quick digression about configuration data: everything that is important is stored in PE resources and encrypted with a Blowfish cipher, using a hard-coded 16-byte key.

Configuration contains the IP address of the C&C, UDP and HTTPS ports, and the name of the system service under which the malware will be installed.

5.1.2 Communication – UDP

It is hard to determine the stealthiness of Send-Safe. If we only care and look for TCP traffic, then it's quite stealthy, but in terms of UDP, it's a whole new story.

UDP is used to determine if the C&C is alive and to register in it. There are various flags and data that can be sent through this channel, but in essence it boils down to the size of the answer. The following C-struct describes the format of the packets sent by the bot:

```
struct req_s {
  BYTE size; /* 72 + size of additional data, itw
  always 72 */
  BYTE req_id; /* always 0x01 */
  BYTE botid[16];
  unk unk_time; /* some strange time related
  structure, in practice always 28 bytes of 0s
  DWORD unk1,unk2; /* always zeros */
  DWORD campaing_id ; /* tends to be 0 */
  struct version {
    WORD ver_hi;
    WORD ver_low;
  } version; /* current: 2,5,0,854 */
  DWORD unk2,unk3; /* always zeroes */
}
```

While the response can contain various flags, in reality what matters is the size of it:

- Eight bytes, C&C is alive but closed: come back some other time
- 24 bytes, C&C is alive and open for businesses: please switch to HTTPS. Every packet is encoded by XORing data with a key derived from the customer id, which in the case of the man1 gang is 'UNREGISTERED'. The following decompiled code shows the algorithm:

```
int __cdecl COMM::xor(_BYTE *a1, int size)
{
  int result; // eax@1
  int i; // [esp+0h] [ebp-18h]@1
  unsigned __int8 v4[16]; // [esp+4h] [ebp-14h]@1

  md5((int)v4, "UNREGISTERED", 12);
  for ( i = 0; i < size; ++i )
  {
    result = v4[i & 0xF] ^ (unsigned __int8)a1[i];
    a1[i] = result;
  }
  return result;
}
```

After encoding, byte 0x02 is added at the beginning. The encoded request is as follows:

```
00000000: 028e 5b6a 4669 76ba 67c0 7a20 a628 26ca
00000010: fdb8 d0c0 5a31 1f66 9f99 365e 0b45 ca69
00000020: 07c6 5b6b 5a31 1f66 9f99 365e 0b45 ca69
00000030: 07c6 5b6b 5a31 1f66 9f99 365e 4b64 9c6a
00000040: 07c6 5b6b 5a31 1f66
```

And the decoded request:

```
00000000: 0248 0001 1c58 69dc f859 4c7e ad6d eca3
00000010: fa7e 8bab 0000 0000 0000 0000 0000 0000
00000020: 0000 0000 0000 0000 0000 0000 0000 0000
00000030: 0000 0000 0000 0000 0000 0000 4021 5603
00000040: 0000 0000 0000 0000
```

5.1.3 Communication – HTTPS

After the malware receives information that the HTTPS port is open, it proceeds to download what's necessary to send spam. The requests performed are rather simple compared to those described previously in this paper. A basic request consists of the C&C address, registered botid and request type. The request type can be:

- 1 – download spam details
- 2 – download target addresses.

Example of HTTPS requests:

```
GET /699206552FDD4E58949E1EC09B199DC6/1 HTTP/1.1
User-Agent: Mozilla
Host: 91.220.131.143:50013
Cache-Control: no-cache
```

In response, the C&C sends a bunch of data, which is basically an encoded zip file:

```
HTTP/1.1 200 OK
Connection: close
Content-Type: text/html
Content-Length: 4618
Server: Indy/9.0.17
## from sslsplit 2017-03-16 19:24:20 UTC
[91.220.131.143]:50013 -> [172.16.15.13]:55201
(4618):
010ltrWjkfb/zpfObS45RPCsZbqUMxH2efmTZsbhyB+9z+y542LP
5lU7jyLl7+JocaCGpwHSNX9I9oV78oU7/OvgZ3jD7CjULL63kq0x
dBSwKi5TSYzuT5wCfKmZxqlZcAaaTujc7ZTSTxGikxE1kxPhTtm3
9hN/
```

One can get a proper zip file using the following python snippet:

```
resp = 'PK\x03\x04' + xor(x[2:].
decode('base64'),md5('UNREGISTERED').digest())
```

This zip file can contain the following files:

- 1 - SMTP details, User-Agent some private key
- 100 - email details, including subject, message body and how to impersonate
- 2 - email addresses of victims

All of the files are additionally wrapped into a simple Type-Length-Value format, which can be parsed with the help of the following Python class:

```
class SFile(M):
  def elem(self):
    size = self.dword()
    flag = self.dword()
    if flag & 0x10000:
      flag ^= 0x10000
      data = self.read(size-8)
    else:
      s = self.dword()
      data = self.read(s+1)
    return flag,data
  def parse(self):
    self.dword()
    cnt = self.dword()
    for i in range(cnt):
      yield self.elem()
```

5.4 Email templates

Like every serious spamming tool, Send-Safe is capable of generating messages based on some sort of template. Although the exact description of how it works is quite complex and is beyond the scope of this paper, what is curious is that one template is used to create content versions for both *Outlook* and other email clients, and the decision as to which one to use is made by a botmaster. Appendix B presents a simple email template captured from the communication of a live sample.

5.5 Curious spamming habits

Send-Safe campaigns are very short lived – each one we observed was active for maximum of two to three days, after which the C&C was completely shut down and the campaign ended. During our research we observed that the C&C is active between around 16:00-21:00 CEST (GMT+02) (by 'active' we mean that it is responding to UDP requests); HTTPS communication starts around 17:30 and continues until 20:30.

6. TOFSEE

Another botnet that we analysed is Tofsee, also known as Gheg [13]. Its main job is to send spam, but it is able to perform other tasks as well. This is possible because of the modular design of this malware: it has one main binary (the one that the user downloads), which later downloads several additional modules from the C&C server – they modify code by overwriting some of the called functions with their own. For example, these modules can, in theory, spread by posting click-bait messages on *Facebook* and *VKontakte* (a Russian social network) – in practice, we haven't observed these modules being used too much.

Communication with the botmaster is implemented using a non-standard protocol built on top of TCP. The first message after establishing the connection is always sent by the server – it contains a mainly random 128-byte key used for

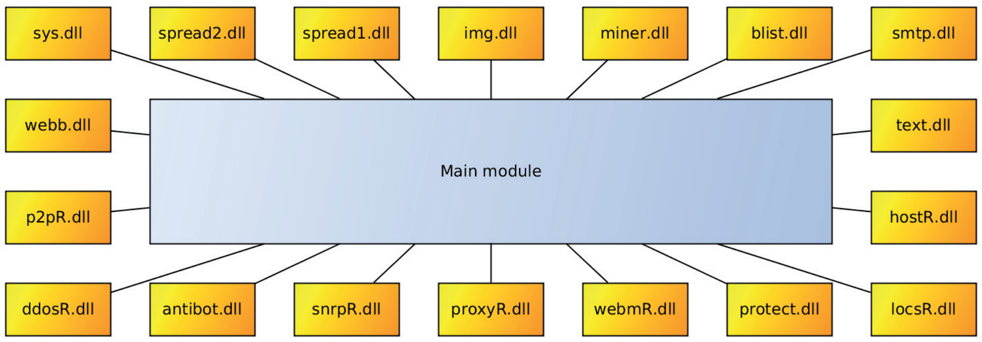

Figure 5: Tofsee modules.

encrypting further communication. Because of this, it is impossible to decode the communication if it wasn't recorded right from its beginning.

Additionally, the bot has a list of resources (in the form of a linked list) in memory. Initially when the bot starts, the list is almost empty and contains only basic information, like bot ID, but it is quickly filled by data received from the server in further messages. Resources can take different forms – for example, it might be a list of mail subjects to be used in spam, but DLL libraries extending bot capabilities are treated as named resources as well. There are a few different resource types – for example, a resource can contain a list of mail subjects to be used in spam, or another DLL, or scripts used for spam, or a list of C&C IP addresses.

The C&C IP list is one of the first messages sent by a server. If, for some reason, the C&C doesn't return its own IP in a C&C list, the connection is terminated and a random server from the newly received list is chosen as the communication partner. This usually happens during connection to one of the C&Cs hard coded in the binary – effectively, they act as a 'pointer' to the real servers.

Sent emails are all randomized – for this purpose, Tofsee uses a dedicated script language. Its body contains macros, which will be replaced randomly by certain strings of characters during parsing – for example, %RND_SMILE will be substituted by one of several emoticons. Thanks to this randomization, simpler spam filters might allow these messages through.

6.1 Technical analysis

The list of C&C IP addresses is hard coded in binary in an encrypted form. The obfuscation algorithm is very simple – it XORs the message with the hard-coded key.

```
def decrypt(s, key, inc):
  result = ""
  parity = 1
  for c in s:
    c = ord(c) ^ key
    result += chr(c)
    key += inc + parity
    key &= 0xFF
    parity = -parity
  return result
```

The data decrypts to a few IP+port pairs – in the analysed sample, the used port was 443 in all of them. The probable reason for this is to conceal communication by using a port dedicated to SSL traffic.

6.2 Communication protocol

The communication protocol is rather simple and is illustrated in Figure 6.

After the client establishes a TCP connection, the server immediately sends a 200-byte-long 'greeting' message – it contains a few useful fields such as:

- encryption key
- public IP of a client
- time on server

This message is 'encrypted' with simple bitwise operations:

```
def greetingXor(data):
  dec=""
  prev=0
  for c in data:
     dec += chr((0xc6 ^ prev ^ ( (ord(c)<<5) |
     (ord(c)>>3) )) & 0xFF)
     prev = ord(c)
return dec, prev
```

Decrypted data forms the following structure:

```
struct greeting {
  uint8_t key[128];
  uint8_t unknown[16];
  uint32_t bot_ip;
  uint32_t srv_time;
  uint8_t reserved[48];
};
```

After this message, everything is encrypted with a 128-byte `key` from the greeting, using the following (probably custom) stream cipher:

```
def xor_crypt(data, key_data):
  res = []
  for c in data:
     it = key_data.it
     key_data.round[it % 7] += key_data.key[it % 128]
     key_data.round[it % 7] &= 0xFF
     res.append(c ^ key_data.round[it % 7])
     key_data.it += 1
  return res
```

Where `key_data.main` is `key` key from the greeting, `key_data.round` is initialized to 'abcdefg', and `it` is initialized to 0.

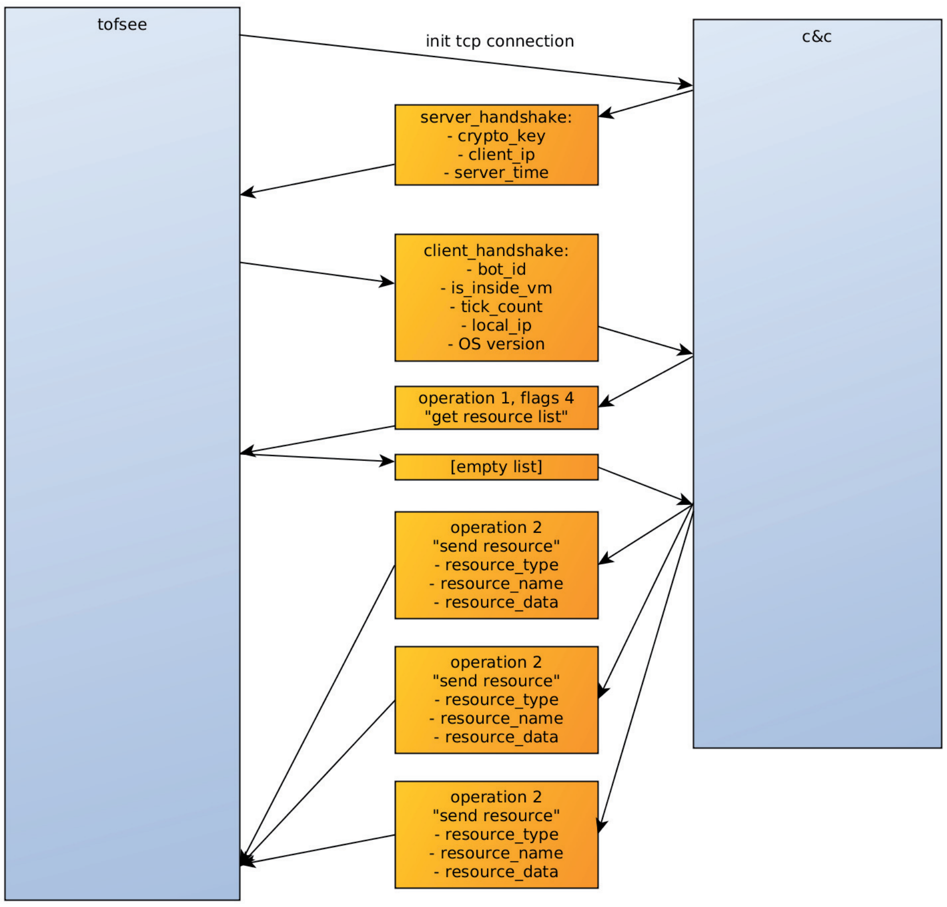

Figure 6: Tofsee comunication.

All messages (with the exception of the greeting message) have a header, represented by the following structure:

```
struct header {
  uint32_t size;
  uint32_t size_decompressed;
  uint32_t checksum_crc32;
  uint8_t flags; // flags & 2 -> is_compressed
  uint32_t op;
  uint32_t subop1;
  uint32_t subop2;
};
```

Data compression is supported by the protocol, but it is only used for bigger messages. The fields op, subop1 and subop2 are certain constants defining the message type – the most important of which is, of course, op. The binary has code for handling a large number of types, but in practice, only a fraction of them are used.

The payload is sent after the header. Its exact structure and contents depend on the message type – some of which will be described in detail below.

The first message sent by the bot has type 1 (op, with subop1 and subop2 being 0) and is a quite big structure:

```
struct botdata {
  uint32_t flags_upd;
  uint64_t bot_id;
  uint32_t unk1;
  uint32_t net_type;
  uint32_t net_flags;
  uint32_t vm_flags;
  uint32_t unk2;
  uint32_t unk3;
  uint32_t lid_file_upd;
  uint32_t ticks;
  uint32_t tick_delta;
  uint32_t born_date;
  uint32_t IP;
  uint32_t unk4;
  uint32_t unk5;
  uint8_t unk6;
  uint8_t operating_system;
  uint8_t unk[46];
};
```

The server response can have different forms as well. The simplest one, op=0, means an empty response (or end of transmission consisting of multiple messages). If op=2, the server sends us a new resource – the message payload in this case has the following structure:

```
struct resource {
  uint32_t type; // Small integer.
  char name[16];
  uint32_t unk;
  uint32_t length;
  uint8_t contents[]; // Size=length.
};
```

6.3 Resources

After the handshake, the server sends a lot of resources – they have the same internal structure.

Resource type	Meaning
1	IP address of C&C or peers (resource name = `work_srv` or `start_srv`)
5	DLL with plug-in – see below (resource name = plug-in name)
8	Local macros, for use during communication
11	Scripts used for spamming
23-40	Config for plug-in (resource name = `img_cfg`, `sys_cfg`, etc.)

Table 7: Resource types and their meanings.

Resources are identified by their type – a small integer (up to 40, but most of them are below 10) and a short name, such as ‘priority’. Some of the most interesting types are described in the following sub-sections.

6.3.1 Type 5

This contains raw plug-in DLL data. Plug-in names are in plaintext in binary data, so we can easily extract plug-in names. As of the time of writing this paper, Tofsee downloads the plug-ins shown in Table 8.

Resource Id	DLL name	DLL MD5 hash
1	ddosR.dll	`fbc7eebe4a56114e55989e50d8d19b5b`
2	antibot.dll	`a3ba755086b75e1b654532d1d097c549`
3	snrpR.dll	`385b09563350897f8c941b47fb199dcb`
4	proxyR.dll	`4a174e770958be3eb5cc2c4a164038af`
5	webmR.dll	`78ee41b097d402849474291214391d34`
6	protect.dll	`624c5469ba44c7eda33a293638260544`
7	locsR.dll	`2d28c116ca0783046732edf4d4079c77`
10	hostR.dll	`c90224a3f8b0ab83fafbac6708b9f834`
11	text.dll	`48ace17c96ae8b30509efcb83a1218b4`
12	smtp.dll	`761e654fb2f47a39b69340c1de181ce0`
13	blist.dll	`e77c0f921ef3ff1c4ef83ea6383b51b9`
14	miner.dll	`47405b40ef8603f24b0e4e2b59b74a8c`
15	img.dll	`e0b0448dc095738ab8eaa89539b66e47`
16	spread1.dll	`227ec327fe7544f04ce07023ebe816d5`
17	spread2.dll	`90a7f97c02d5f15801f7449cdf35cd2d`
18	sys.dll	`70dbbaba56a58775658d74cdddc56d05`
19	webb.dll	`8a3d2ae32b894624b090ff7a36da2db4`
20	p2pR.dll	`e0061dce024cca457457d217c9905358`

Table 8: Plug-ins downloaded by Tofsee.

Looking at the names, it’s clear that apart from spamming, Tofsee also has a lot of other functions – like coordinated DDoS, cryptocurrency mining, or spreading via various channels. We’ll skip detailed analysis of those modules here, but those interested can read our longer article on the topic published on the cert.pl blog [14].

6.3.2 Type 11

This type contains periodically updated scripts in a dedicated language, which are used to send spam. An example script can be found in Appendix C.

Since some of the variables need to contain a literal newline character, several macros are hard coded in binary for that very purpose, for example, %SYS_N.

6.3.3 Type 8

This chunk contains local macros. Because different email scripts sometimes use macros with the same name but different content, macros can be local. The resource names are of `NUM%VAR` form, for example, `1910%TO_NAME`, where `1910` is a number of the script being the scope of macro `%TO_NAME`.

Variable substitutions can be recursive, so expanded macros can be expanded further. The script language also allows for more complicated constructs, such as `%RND_DIGIT[3]`, meaning three random digits (often used in the hexadecimal description of colour), or `%{%RND_DEXL}{ %RND_SMILE}{}`, meaning a random choice between `%RND_DEXL`, `%RND_SMILE` and an empty string. As we can see the language is quite flexible.

6.3.4 Type 23-40

These chunks contain config of some plug-ins. All values are named by human-readable keys, and parsing this config is trivial:

```
def parse_config(payload):
  log.info(chunks(payload.split('\x00'), 2))
  return dict(chunks(payload.split('\x00'), 2)[:-1])
```

6.4 Tracking

With this knowledge about Tofsee’s protocol, we can start to track it automatically, which we’ve been doing since December 2016. During this time, we have collected 29 unique configs from the C&C server.

Tofsee development is rather irregular – sometimes as many as four updates per day are released, but between them, there are long periods of inactivity.

A small slowdown can be observed during January 2017, which may be related to DGA sinkholing performed by Swiss CERT [15] during that time.

Looking at the updates, it’s obvious that botnet operators care about some functionalities more than others. For example, while the botnet miner is still being sent to every victim, the IP address of the gate is long since dead. Either the botnet owners don’t care about that, or they don’t even know about it. Similarly, while the `spread` plug-in gets updates sometimes, it’s updated not nearly often enough, and the IPs it references are long since dead. In contrast, the C&C and work server addresses and `psmtp_cfg` plug-in are always up to date – because those are necessary for uninterrupted spam operations.

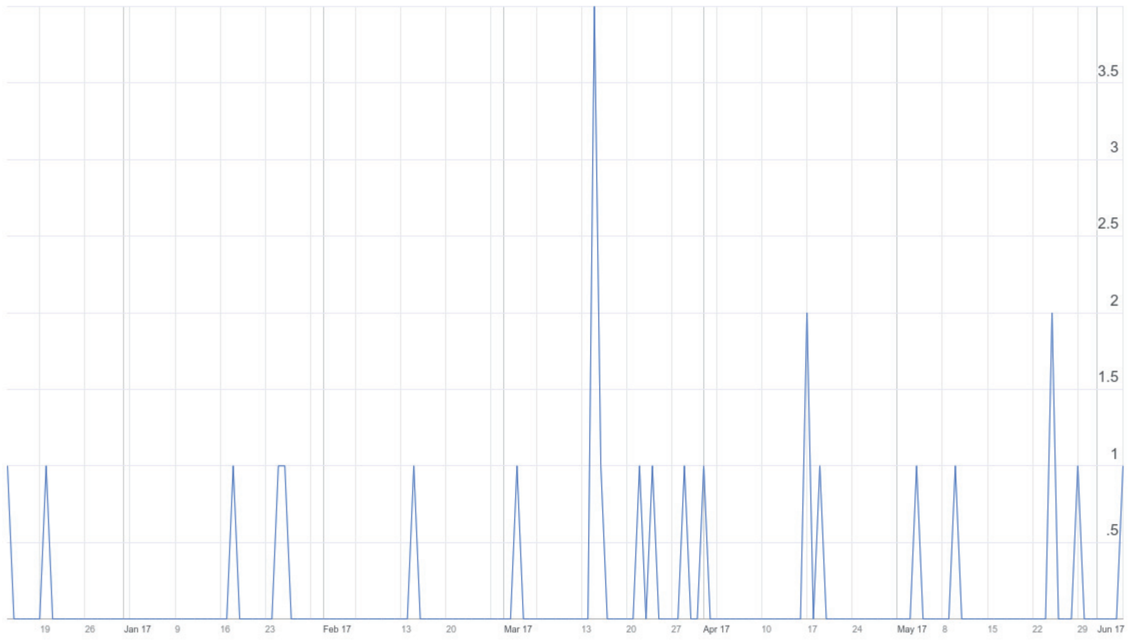

Figure 7: Tracking Tofsee.

7. CLOSING WORDS

We hope that with this paper we manage to lower the entrypoint for monitoring spam botnets. We think that spam botnets are really interesting targets for analysis and a still growing threat, especially in the twilight of the EK era, and its important that we expand our visibility of their operations and eventually stop the perpetrators.

ACKNOWLEDGEMENTS

The authors would like to thank the following for their help: Adam Krasuski, Paul Burbage, Matthew Mesa, J of Techhelplist, and Pawel Srokosz.

REFERENCES

[1] Shulmin, A. The banking trojan emotet: Detailed analysis. 2015. https://securelist.com/69560/thebanking-trojan-emotet-detailed-analysis.

[2] Salvio, J. New banking malware uses network sniffing for data theft. 2014. http://blog.trendmicro.com/trendlabs-security-intelligence/new-banking-malware-uses-network-sniffing-for-data-theft/.

[3] Techhelplist, Emotet. https://techhelplist.com/component/tags/tag/190-emotet.

[4] Srokosz, P. Analysis of emotet v4. 2017. https://www.cert.pl/en/news/single/analysisof-emotet-v4.

[5] FBI, Application for a search warrant – procedure to disrupt the kelihos botnet. 2017. https://www.justice.gov/opa/press-release/file/956521/download.

[6] Yang, K. Dissecting latest kelihos peer exchange communication. 2013. https://blog.fortinet.com/2013/07/18/dissecting-latest-kelihos-peer-exchangecommunication.

[7] Krasuski, A. Necurs – hybrid spam botnet. 2016. https://www.cert.pl/en/news/single/necurshybrid-spam-btnet/.

[8] Sofia.luis. Necurs proxy module with DDoS features. http://blog.anubisnetworks.com/blog/necurs-proxy-module-with-ddos-features?hs_preview=qztxAuUY-4862144940.

[9] Reaves, J. Me and Mr. robot: Tracking the actor behind the man1 crypter. 2016. https://www.fidelissecurity.com/threatgeek/2016/07/me-and-mrrobot-tracking-actor-behind-man1-crypter.

[10] Spamhaus. ROKSO: Ruslan ibragimov/send-safe.com. https://www.spamhaus.org/rokso/spammer/SPM672/send-safe.com.

[11] VirusTotal. Sample - 72b1508d72b43b9322676efb06210aaa - sse.exe. https://www.virustotal.com/en/file/f64bef872fa1488bc3dd9f09a46c5cba b6ce338ddc5350effb8daaa4175fc50f/analysis/.

[12] Dr.Web, Drwtoday.vdb , 05 apr 2016, 16:40 gmt-8. http://live.drweb.com/base_info/?id=262405.

[13] Krasuski, A. Tofsee – modular spambot. 2016. https://www.cert.pl/en/news/single/tofseeen/.

[14] Jedynak, J. Tofsee – modules description. 2017. https://www.cert.pl/en/news/single/tofseemodules/.

[15] GovCERT.ch. Tofsee spambot features .ch dga – reversal and countermesaures. 2016. https://www.govcert.admin.ch/blog/26/tofsee-spambot-features-.chdga-reversal-and-countermesaures.

HASHES (SHA256)

Emotet’s spam module: 0eb2eb8c5c21cfd6b89c1e14b3b66f8 69148f06fa0611ad3e7aa06e285a7e9c6

Kelihos binary: 7336b25d9c3389867e159e89f88e2d9f58c31c3a141806efec3e5c5cf0cc202f

Necurs main module: 2cf6ba0346b92192bcf4941b3864df23e01c65e7e37cfe4648a72fe5d1e0c848

Necurs main module: c54d3cef68932a72c8ce3194f2672c1396bf5fedf5dfc61aed3ccdb8b4feca8a

Necurs rootkit (2017-04-05): c4b4f8bc15b08c5bf937660125d436ebaa92ad702d207d4afd57db0bec45a34c

Send-Safe: 68ce6a73e5eb1e538eb21a63a613761feb259e6eae55bf1022ab3f86fbbbeac1

Tofsee main module: deed28bc0060e5fd712c8b495dd6a992d417e014a78539a4eb32c2a680e69b2a

APPENDIX A: KELIHOS ENCRYPTION ALGORITHMS

Appendix A1: bitcrypt1

```
def bitstuff_enc(b):
  chr = b
  mask = 1
  prevbit = chr & 1
  out = 0
  for i in range(8):
    lobit = chr & 1
    if lobit != prevbit:
      out |= mask
    chr >>= 1
    if i:
      mask <<= 1
    prevbit = lobit
  out |= mask
  if b & 1 == 0:
      out = ~out
  return out & 0xFF

def bit1_enc(string):
  return map_string(string, bitstuff_enc)
```

Appendix A2: bitcrypt2

```
def bitstuff2_enc(chr):
  result = 0
  mask = chr & 1
  prevbit = mask == 0
  for i in range(8):
    curbit = chr & 1
    if curbit == prevbit:
      mask ^= 1
    if mask:
      result |= 1
    chr >>= 1
    result <<= 1
    prevbit = curbit
  return result >> 1

def bit2_enc(string):
  return map_string(string, bitstuff2_enc)
```

Appendix A3: mathops crypt

```
void crypt_mathops(int a1, string *data) {
  for (int i = 0; i < data->length; i++) {
    chr = getCharAtPos(data, i);
    if ( *chr & 1 )
      enc = 32 * (*chr & 0xFE) | (*chr >> 2) & 0x3E;
    else
      enc = 4 * *chr | (*chr >> 5) & 6 | 1;
    *chr = enc;
  }
}
```

Appendix A4: Full encryption routine

```
def decrypt_raw_data(data):
  header = data[:24]
  header = fst_block[:4] + visdecrypt2(bit2_dec(fst_
  block))[4:24] # decrypt header
  rnd_seed = int32(header[0:4])
  blk1len_enc = int32(header[4:8])
  blk2len_enc = int32(header[8:12])
  blk3len_enc = int32(header[12:16])
  pkttype_enc = int32(header[16:20])
  unk2len_enc = int32(header[20:24])

  blk1len = blk1len_enc - rnd_seed
  blk2len = blk2len_enc - blk1len_enc
  blk3len = blk3len_enc - blk2len_enc
  unk2len = unk2len_enc - blk3len_enc - 95
  pkttype = pkttype_enc - blk1len_enc - 197

  print 'rndseed: [{:8x}] => {:x}'.format(rnd_seed,
  rnd_seed)
  print 'block 1 len: [{:8x}] => {:x}'.
  format(blk1len_enc, blk1len)
  print 'block 2 len: [{:8x}] => {:x}'.
  format(blk2len_enc, blk2len)
  print 'block 3 len: [{:8x}] => {:x}'.
  format(blk3len_enc, blk3len)
  print 'pkttype: [{:8x}] => {:} '.format(pkttype_
  enc, pkttype)
  print 'unk2len: [{:8x}] => {:x}'.format(unk2len_
  enc, unk2len)

  data_strings_enc = data[24:]
  data_strings = bit1_dec(data_strings_enc)
  blk1 = data_strings[:blk1len] # random bytes and
  decryption algorithm seed
  blk2 = data_strings[blk1len:blk1len+blk2len] #
  encrypted blowfish key
  blk3 = data_
  strings[blk1len+blk2len:blk1len+blk2len+blk3len] #
  encrypted peer list

  print 'block 1:', blk1.encode('hex')
  print 'block 2:', blk2.encode('hex')
  print 'block 3:', blk3.encode('hex')

  blk1 = visdecrypt2(blk1)
  blk1 = bit1_dec(blk1)
  saltparam = ord(blk1[0]) # length of salt is
  encoded in first byte of block
  lonib = saltparam & 0xF
  hinib = (saltparam >> 4) & 0xF

  real_blk1 = blk1[lonib+hinib+1:]
  print 'real block 1:', real_blk1.encode('hex')
```

APPENDIX B: SEND-SAFE EMAIL TEMPLATE

```
Message-ID: {%MSGID%}
{%NOT_OUTLOOK%}Date: {%DATE%}
{%RANDOMLY%}Reply-To: {%FROM%}
From: {%FROM%}
{%NOT_OUTLOOK%}{%XMAILER_HEADER%}
{%NOT_OUTLOOK%}{%RANDOMLY%}X-Accept-Language: en-us
{%NOT_OUTLOOK%}MIME-Version: 1.0
{%TOCC_HEADERS%}
Subject: {%SUBJECT%}
{%OUTLOOK%}Date: {%DATE%}
{%OUTLOOK%}MIME-Version: 1.0
{%OUTLOOK%}Content-Type: multipart/alternative;
{%OUTLOOK%} boundary="{%BOUNDARY1%}"
{%NOT_OUTLOOK%}Content-Type: text/html;
{%NOT_OUTLOOK%} charset="utf-8"
{%NOT_OUTLOOK%}Content-Transfer-Encoding: 7bit
{%OUTLOOK%}{%RANDOMLY%}X-Priority: 3
{%OUTLOOK%}X-MSMail-Priority: Normal
{%OUTLOOK%}{%XMAILER_HEADER%}
{%OUTLOOK%}X-MimeOLE: Produced By Microsoft MimeOLE
V{%MIMEOLE_VERSION%}

{%OUTLOOK%}This is a multi-part message in MIME
format.
{%OUTLOOK%}
{%OUTLOOK%}--{%BOUNDARY1%}
{%OUTLOOK%}Content-Type: text/plain;
{%OUTLOOK%} charset="utf-8"
{%OUTLOOK%}Content-Transfer-Encoding: quoted-
printable
{%OUTLOOK%}
{%OUTLOOK%}{%BEGIN_QUOTEDPRINTABLE%}
{%OUTLOOK%}{%PLAINTEXT_MSG%}
{%OUTLOOK%}{%END_QUOTEDPRINTABLE%}
{%OUTLOOK%}--{%BOUNDARY1%}
{%OUTLOOK%}Content-Type: text/html;
{%OUTLOOK%} charset="utf-8"
{%OUTLOOK%}Content-Transfer-Encoding: quoted-
printable
{%OUTLOOK%}
{%OUTLOOK%}{%BEGIN_QUOTEDPRINTABLE%}
{%NOT_OUTLOOK%}{%BEGIN_SPLIT76%}
{%BEGIN_PLAINTEXT_SRC%}
<html>
<p><img src="http://www.entwistle-law.com/images/
logo.png" alt="" width="598" height="33" /></35
<p>My name is Vincent Cappucci and I am a
senior partner at ENTWISTLE &amp; CAPPUCCI
LLP.<br <p><a href="http://fortyfour.jp/divorce/
divorce.php?id={%BEGIN_BASE64%}{%EMAIL%}{%END_
BASE64%}"><p>Thank you<br />Vincent R. Cappucci<br
/>Senior Partner<br />{%FROMEMAIL%}<br />Phone: 212-
</html>
{%END_PLAINTEXT_SRC%}
{%OUTLOOK%}{%END_QUOTEDPRINTABLE%}
{%NOT_OUTLOOK%}{%END_SPLIT76%}
{%OUTLOOK%}
{%OUTLOOK%}--{%BOUNDARY1%}--
```

APPENDIX C: TOFSEE TYPE 11 SCRIPT

```
From: "%NAME" <%FROM_EMAIL>
To: %TO_EMAIL
Subject: %SUBJ
Date: %DATE
MIME-Version: 1.0
Content-Type: multipart/mixed;
boundary="%BOUNDARY1"

--%BOUNDARY1
Content-Type: multipart/alternative;
boundary="%BOUNDARY2"

--%BOUNDARY2
Content-Type: text/plain;
charset="%CHARSET"
Content-Transfer-Encoding: quoted-printable

{qp1-}%GI_SLAWIK{/qp}

--%BOUNDARY2
Content-Type: text/html;
charset="%CHARSET"
Content-Transfer-Encoding: quoted-printable

{qp0+}%GI_SLAWIK{/qp}

--%BOUNDARY2--
--%BOUNDARY1
Content-Type: application/zip;
name="%ATTNAME1.zip"
Content-Transfer-Encoding: base64
Content-Disposition: attachment;
filename="%ATTNAME1.zip"
%JS_EXPLOIT

--%BOUNDARY1--
- GmMxSend
v SRV alt__M(%RND_NUM[1-4])__.gmail-smtp-
in.l.google.com
U L_SKIP_5 5 __M(%RND_NUM[1-5])__
v SRV gmail-smtp-in.l.google.com
L L_SKIP_5
C __v(SRV)__:25
RS
mx_smtp_01.txt
o ^2
m %FROM_DOMAIN __A(4|__M(%HOSTS)__)__
W """EHLO __A(3|__M(%{mail}{smtp}%RND_NUM[1-
4].%FROM_DOMAIN)__)__\r\n"""
RS
mx_smtp_02.txt
o ^2 ^3
L L_NEXT_BODY
v MI 0
- m %FROM_EMAIL __M(%FROM_USER)__@__M(%FROM_
DOMAIN)__
W """MAIL From:<__M(%FROM_EMAIL)__>\r\n"""
RS
mx_smtp_03.txt
I L_QUIT ^421
o ^2 ^3
L L_NEXT_EMAIL
U L_NO_MORE_EMAILS @ __S(TO|__v(MI)__)__
W """RCPT To:<__l(__S(TO|__v(MI)__)__)__>\r\n"""
```

```
RS
mx_smtp_04.txt
I L_OTLUP ^550
I L_TOO_MANY_RECIP ^452
o ^2 ^3
v MI __A(1|__v(MI)__,+,1)__
u L_NEXT_EMAIL 1 __A(1|__v(MI)__,<,1)__ L L_NO_MORE_
EMAILS u L_NOEMAILS 0 __A(1|__v(MI)__,>,W """DATA\
r\n"""
RS
mx_smtp_05.txt
o ^2 ^3
m %SS1970H __P(__t(126230445)__|16)__
m %TO_EMAIL """<__l(__S(TO|0)__)__>"""
m %TO_NAME __S(TONAME|0)__
W """__S(BODY)__\r\n.\r\n"""
RS
mx_smtp_06.txt
I L_SPAM ^550
o ^2 ^3
+ m
H TO -1 OK
J L_NEXT_BODY
L L_OTLUP
+ h
h """Delivery to the following recipients failed.
__l(__S(TO|__v(MI)__)__)__"""
H TO __v(MI)__ HARD
J L_NEXT_EMAIL
L L_TOO_MANY_RECIP
H TO __v(MI)__ FREE
J L_NO_MORE_EMAILS
L L_QUIT
W """QUIT\r\n"""
RS
mx_smtp_07.txt
o ^2 ^3
L L_NOEMAILS
E 1
L L_SPAM
+ A
H TO -1 FREE
o ^2 ^3
```