Tofsee – modular spambot

Archived: 2026-04-05 16:21:15 UTC

Tofsee, also known as Gheg, is another botnet analyzed by CERT Polska. Its main job is to send spam, but it is able to do other tasks as well. It is possible thanks to the modular design of this malware – it consists of the main binary (the one user downloads and infects with), which later downloads several additional modules from the C2 server – they modify code by overwriting some of the called functions with their own. An example of some actions these modules perform is spreading by posting click-bait messages on Facebook and VKontakte (Russian social network).

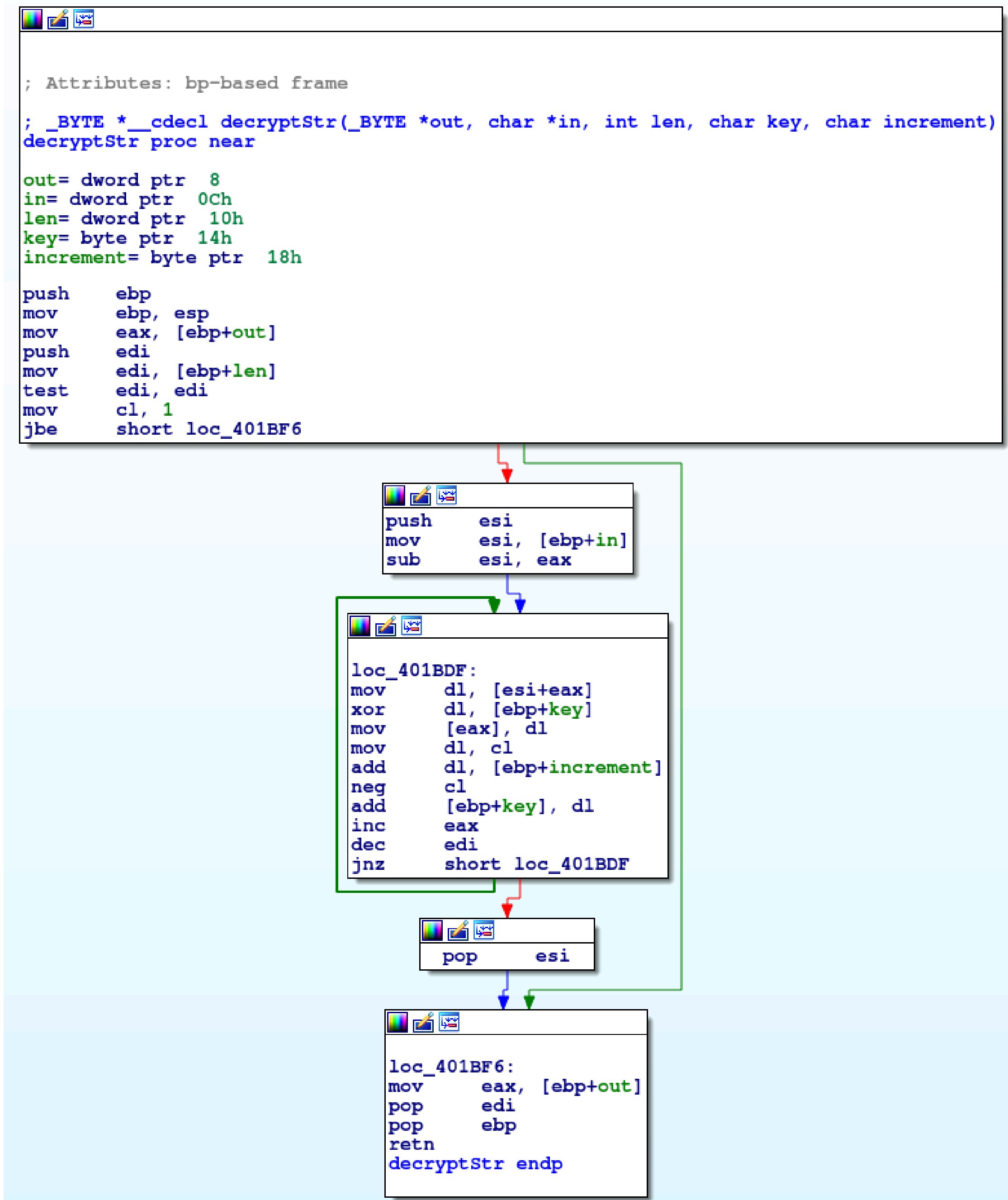
Bot communicates with the botmaster using non-standard protocol built on top of TCP. The first message after establishing the connection is always sent by the server – the most important thing it contains is a random 128-byte key used for encrypting further communication. It is therefore impossible to decode the communication if one wasn't listening right from its beginning.

At all times, bot keeps a list of resources (in the form of linked list) in memory. Just after bot starts, the list is almost empty and contains only basic information, such as 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. Additionally, one of resources – *work_srv* – contains a list of C2 IP addresses. It is one of the first messages sent by server and, interestingly, may not contain itself – in this case, connection is soon terminated and a random server from the newly received list is chosen as communication partner. This usually happens during connection to one of C2s hardcoded in the binary – effectively, they act as "pointer" to real servers.

All sent emails are randomized – for this purpose, Tofsee uses a special script language (an example file is in technical analysis section). 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 pass these messages through.

Technical analysis

The C2 IP address list is hardcoded in the binary in an encrypted form. The algorithm used for obfuscation is very simple – it XORs the message with the hardcoded key.



Data decrypts to three IP+port pairs – at least in the analyzed sample, the port was equal to 443 for all of them. The probable reason is to conceal communication by using port dedicated for SSL traffic.

Communication protocol

After establishing TCP connection the first message is sent by the server. Its size is always 200 bytes long (though not all bytes are used – the final ones seem to be reserved, perhaps for future expansion of the protocol). This message is also obfuscated by simple bitwise operations:

	def greetingXor(data):
	dec=""

```
res=198
for c in data:
dec+=chr((res^(32*ord(c)|(ord(c)>>3)))&0xFF)
res=ord(c)^0xc6
return dec, res
```

Decrypted data form the following structure (we were not able to find the meaning of some of the fields):

```
struct greeting{
uint8_t key[128];
uint8_t unk1[16];
uint32_t bot_IP;
uint32_t srv_time;
uint8_t unk2[48];
};
```

From this point on all traffic (both incoming and outgoing) is encrypted using the the 128-byte key received in the first message. The key is modified after every sent or received byte, so it is impossible to decrypt the transmission without listening to it from the beginning. XORing is used in such a way, that a single function can both encrypt and decrypt messages:

```
def xorStream(data, key, main_key, it):
res=""
for c in data:
key[it%7]+=main_key[it%128]
key[it%7]&=0xFF
res+=chr(ord(c)^(key[it%7]))
it+=1
return res
```

Parameters:

- *data* – raw data
- *key* – short, 7-byte key, initialized by “abcdefg” bytes before the first message
- *main_key* – 128-byte key from the greeting message
- *it* – number of bytes sent/received till now

All messages (except the greeting) consist of header and payload. The header is represented by the following structure:

The protocol supports data compression, but it is only used for bigger messages. Fields *op*, *subop1* and *subop2* are certain constants defining message type. The binary contains code handling a large number of types, but in practice, only a fraction of them is used.

Payload is sent after the header. Its exact structure and contents depend on message type – some of them will be described in details below.

The first message sent by the bot has types {1,0,0} (*op*, *subop1* and *subop2*, respectively) and is a quite big structure:

	struct botdata{
	uint32_t flags_upd;
	uint64_t botID;
	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 OS;
	uint8_t unk[46];
	};

Some of the field names (such as *lid_file_upd*) we got for free – we did not have to reverse them by analyzing their usage, since bot saved them under those exact indices to internal data structure, mapping variable names to their contents.

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 is in this case of the following structure:

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

Usually after connecting to C2 server hardcoded in the binary the first message (after greeting) received by the bot is a single resource named *work_srv*, which contains a list of a couple of IP addresses and ports (this time different than 443), on which true C2 servers are listening. The bot then disconnects from the current server and, after a while, starts the communication over with one of the freshly obtained IPs.

If *op*=1, the message's meaning depends on *subop2* and, additionally, first four bytes of payload (which are apparently used as flags in this case). For example, if these conditions are met: *op*=1, *subop2*&1=0, *flags*=4, the message is a C2 request for all resources the bot has. The bot's response is then a concatenated list of resources in a form similar to the showed above, after which server sends tens or hundreds type 2 messages (containing resources) – resources, which bot does not have yet.

Resources

Every resource is identified by its 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 include:

Type 5

Contains plugin DLLs. Since they don't have all of their symbols stripped, we could quickly guess plugins' tasks. As of today, Tofsee downloads the following plugins:

Name of resource – number	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

Judging by these names, apart from spamming, Tofsee also has other functions, such as coordinated DDoS, or cryptocurrency mining (as it turns out, one of the resources being downloaded is a Litecoin miner).

Type 11

Contains periodically updated scripts in an atypical language, which are used to send spam. Example 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
R
S 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"""
R
S 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"""
R
S mx_smtp_03.txt
```

<table>
<tr><td></td><td>I L_QUIT ^421</td></tr>
<tr><td></td><td>o ^2 ^3</td></tr>
<tr><td></td><td>L L_NEXT_EMAIL</td></tr>
<tr><td></td><td>U L_NO_MORE_EMAILS @ __S(TO|__v(MI)__)__</td></tr>
<tr><td></td><td>W """RCPT To:<__l(__S(TO|__v(MI)__)__)__>\r\n"""</td></tr>
<tr><td></td><td>R</td></tr>
<tr><td></td><td>S mx_smtp_04.txt</td></tr>
<tr><td></td><td>I L_OTLUP ^550</td></tr>
<tr><td></td><td>I L_TOO_MANY_RECIP ^452</td></tr>
<tr><td></td><td>o ^2 ^3</td></tr>
<tr><td></td><td>v MI __A(1|__v(MI)__,+,1)__</td></tr>
<tr><td></td><td>u L_NEXT_EMAIL 1 __A(1|__v(MI)__,<,1)__ L L_NO_MORE_EMAILS u L_NOEMAILS 0 __A(1|__v(MI)__,>,0)__</td></tr>
<tr><td></td><td>W """DATA\r\n"""</td></tr>
<tr><td></td><td>R</td></tr>
<tr><td></td><td>S mx_smtp_05.txt</td></tr>
<tr><td></td><td>o ^2 ^3</td></tr>
<tr><td></td><td>m %SS1970H __P(__t(126230445)__|16)__</td></tr>
<tr><td></td><td>m %TO_EMAIL """<__l(__S(TO|0)__)__>"""</td></tr>
<tr><td></td><td>m %TO_NAME __S(TONAME|0)__</td></tr>
<tr><td></td><td>W """__S(BODY)__\r\n.\r\n"""</td></tr>
<tr><td></td><td>R</td></tr>
<tr><td></td><td>S mx_smtp_06.txt</td></tr>
<tr><td></td><td>I L_SPAM ^550</td></tr>
<tr><td></td><td>o ^2 ^3</td></tr>
<tr><td></td><td>+ m</td></tr>
<tr><td></td><td>H TO -1 OK</td></tr>
</table>

	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"""
	R
	S mx_smtp_07.txt
	o ^2 ^3
	L L_NOEMAILS
	E 1
	L L_SPAM
	+ A
	H TO -1 FREE
	o ^2 ^3

The language is slightly similar to assembly – for example “J” as the first character in line means *jump*, and “L” – defines *label*. Script contains macros, which are substituted into other text at runtime – for example *%ATTNAME1*.

Type 7

Contains general purpose macros. The name of these resources is the same as macro’s they describe, for example *%DATE_RAN_SUB* (likely abbreviation of *DATE RANDOM SUBJECT*). The resource content is a newline-separated list of substitutions, for example:

	%NAME posted something on your wall
	Newsletter from %NAME
	%NAME changed her status
	User %NAME is available to chat
	%NAME answered your message
	New message from %NAME
	%NAME requested to be your friends
	User %NAME sent you a message
	%NAME likes your status
	Do you know %NAME?
	%NAME sent you invitation
	New friendship request from %NAME
	%NAME is your friend now

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

Type 8

Contains local macros. Since different email scripts might want to use macros with the same name, but different content, some of the macros are local. The resource names are of *NUM%VAR* form, for example *1819%TO_NAME*, where 1819 is number of the script being the scope of macro *%TO_NAME*.

Variable substitutions are recursive, as seen on aforementioned example of *%DATE_RAN_SUB* – macros can contain other macros. The script language also allows for more complicated constructs, such as *%RND_DIGIT[3]*, meaning three random digits (often used in color's hexadecimal description), 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.

Rest of the types contain only a handful of resources and are less interesting from our perspective, so we will skip their description in this article.

Hash of the analyzed binary and YARA rules matching this malware family will conclude this analysis.

Hash:
ae0d32e51f36ce6e6e8c5ccdc3d253a0 - analyzed sample (main binary, before unpacking)

YARA rules:

	rule tofsee
	{
	meta:
	author="akrasuski1"
	strings:
	$decryptStr = {32 55 14 88 10 8A D1 02 55 18 F6 D9 00 55 14}
	$xorGreet = {C1 EB 03 C0 E1 05 0A D9 32 DA 34 C6 88 1E}
	$xorCrypt = {F7 FB 8A 44 0A 04 30 06 FF 41 0C}
	$string_res1 = "loader_id"
	$string_res2 = "born_date"
	$string_res3 = "work_srv"
	$string_res4 = "flags_upd"
	$string_res5 = "lid_file_upd"
	$string_res6 = "localcfg"
	$string_var0 = "%RND_NUM"
	$string_var1 = "%SYS_JR"
	$string_var2 = "%SYS_N"
	$string_var3 = "%SYS_RN"
	$string_var4 = "%RND_SPACE"
	$string_var5 = "%RND_DIGIT"
	$string_var6 = "%RND_HEX"
	$string_var7 = "%RND_hex"
	$string_var8 = "%RND_char"

	$string_var9 = "%RND_CHAR"
	condition:
	(7 of ($string_var*) and 4 of ($string_res*))
	or
	(7 of ($string_var*) and 2 of ($decryptStr, $xorGreet, $xorCrypt))
	or
	(4 of ($string_res*) and 2 of ($decryptStr, $xorGreet, $xorCrypt))
	}

Source: https://www.cert.pl/en/news/single/tofsee-en/