Botnet analysis PictureXX.JPG

Table of Contents

Few words by the author

Month ago, a friend of mine (ph0ton) send me a malware sample that he catch in the wild. At the time I quickly check it out and saw what it basically does, then put it in the “TODO” box, realizing that I will NEVER EVER open it again.

However, few days ago I was really bored yet inspired, so I decide to write a few words about this one, since as far as I can see it's still fresh... and also another friend of mine catch it.

Well that's the story about this material. If you were expecting something smart or touching here – sorry! Better luck next time.

I. Getting infected

The infection method it best known as "exploiting human stupidity", therefore it's targeting the basic users, which of course are the masses.
The only way someone can get infected is to manually start the malware on his machine, so unless you execute it, you are OK.

Spreading is designed as a misleading message including download URL, transferred on the instant messengers ICQ, SkyPe, GTalk, Pidgin, AIM, MSN and YIM, and Facebook's web chat.

The sample I got is named "*Picture11.JPG_www.facebook.com*". It's 135 kilobytes big, and its MD5 checksum is AC165F6E6E32605DB2226C6FAA086F89.
The extension seems to be ".com", but is actually a standard 32bit MZ-PE executable written in C.
It does not contain any TLS table, and has an intact compile stamp saying "*04/03/2012*".

So 4th of March is most probably its spread date.

The malware is structured by two parts:

- a loader that decrypt and drops the payload;
- and the actual payload that do the malicious job;

II. Protections and encryptions

1. Hidden API functions

Crucial API functions used are defined as checksum DWORD values inside the loader, and then fetched out directly from the PEB data.
For example, if "*SetThreadContext*" is needed, his checksum will be generated based on the function name like so:

```
char apiName[] = "SetThreadContext";
int i, crc = 0;
for(i = 0; i < strlen(apiName); i++) {
   crc = _lrotr(crc,4)+apiName[i];
}
```

The resulting "crc" value of "SetThreadContext" is 0xB590E453.
So that value 0xB590E453 is hardcoded into the loader's body, and later matched against the produced CRC of Kernel32's enumerated API functions.
This trick is used only by the loader. The payload's API functions are not hidden.

2. XOR encryption over the payload

The payload is actually fully functional executable, that is stored XOR encrypted in the ".data" section of the loader. The XOR encryption uses "key" value, that may vary in the different malware versions, therefore bypassing any basic fingerprinting from the AV vendors.

Here's the decrypt algorithm used:

```
unsigned char data[] = "..."; // Data to decrypt
unsigned char magic[] = "\x2C\x8C\x94\x9E\x03\x75\x37\x25"\
                        "\x94\xCC\xB9\x32\xC6\x4C\xD7\xDB"\
                        "\x5B\x37\x62\x6F\x33\x0D\x44\x36"\
                        "\x95\xF5\x2F\xBA\x46\x3A\xD9\x38";
char matrix[256];
int i, a = 0, b, ECX = 0x100, EDX;
// Init the decoder matrix
for(i = 0; i < 0x100; i++) {
   matrix[i] = i;
}
// Shuffle the matrix
for(i = 0; i < 0x100; i++) {
   a = (a+(matrix[i]+magic[i%32]))&0xFF;
   b = matrix[a];
   matrix[a] = matrix[i];
   matrix[i] = b;
}
// Decode the data
for(i = 0; i < 144; i++) {
   ECX = (ECX+1)&0xFF;
   EDX = matrix[ECX];
   a = (a+EDX)&0xFF;
   matrix[ECX] = matrix[a];
   matrix[a] = EDX;
   data[i] = data[i]^matrix[(matrix[ECX]+EDX)&0xFF];
}
```

At the end, "data" holds a fully functional application, that can be executed separately.

3. Self code injection

The payload code is injected in previously executed (and suspended) process of the same application that is later resumed, after its memory gets rewritten.

4. Encoded strings

At first glance the payload executable doesn't hold any interesting strings, except those put from the compiler. The reason is because, everything crucial is "encoded" with a character replacing algorithm.

For example, let's take the first decoded string, that turns out to be the Mutex name of the "protector" - "\x5E\x08\x16\x04\x09\x0C\x09\x0E\x05\x25\x5C\x04\x09\x02..."

Taken as a string, it's meaningless and wont be recognized by the disassembler / debugger until it gets decoded.

So here's how this string gets decoded:

```
char s[] = "\x5E\x08\x16\x04\x09\x0C\x09\x0E\x05\x25\x5C\x04\x09\x02\x0C\x03"\
   "\x04\x25\x47\x19\x0F\x08\x19\x03"; // String to decode
char t[] = "\x71\x77\x65\x72\x74\x79\x75\x69\x6F\x70\x61\x73\x64\x66\x67\x68"\
   "\x6A\x6B\x6C\x7A\x78\x63\x76\x62\x6E\x6D\x31\x32\x33\x34\x35\x36"\
   "\x37\x38\x39\x30\x20\x2E\x3D\x3F\x7E\x21\x40\x23\x24\x25\x5E\x26"\
   "\x2A\x28\x29\x5F\x2D\x2B\x3A\x3B\x5C\x2F\x7C\x2C\x5D\x5B\x7B\x7D"\
   "\x3E\x3C\x22\x60\x51\x57\x45\x52\x54\x59\x55\x49\x4F\x50\x41\x53"\
   "\x44\x46\x47\x48\x4A\x4B\x4C\x5A\x58\x43\x56\x42\x4E\x4D\x27";
int i;
for(i = 0; i < strlen(s); i++) {
   s[i] = t[s[i]-0x01];
}
printf("Decrypted as: \"%s\"\n", s);
```

The result is now an ASCII valid string "Microsoft Browser Engine".

Since the algorithm is two way encoding, it can be not only decoded, but also encoded, like so:

```
char s[] = "Microsoft Browser Engine"; // String to encode
char t[] = "\x25\x2A\x43\x2C\x2D\x2E\x30\x5F\x32\x33\x31\x36\x3C\x35\x26\x3A"\
   "\x24\x1B\x1C\x1D\x1E\x1F\x20\x21\x22\x23\x37\x38\x42\x27\x41\x28"\
   "\x2B\x4F\x5C\x5A\x51\x47\x52\x53\x54\x4C\x55\x56\x57\x5E\x5D\x4D"\
   "\x4E\x45\x48\x50\x49\x4B\x5B\x46\x59\x4A\x58\x3E\x39\x3D\x2F\x34"\
   "\x44\x0B\x18\x16\x0D\x03\x0E\x0F\x10\x08\x11\x12\x13\x1A\x19\x09"\
   "\x0A\x01\x04\x0C\x05\x07\x17\x02\x15\x06\x14\x3F\x3B\x40\x29";
int i;
printf("Encrypted as: \"");
for(i = 0; i < strlen(s); i++) {
   printf("\\x%02X", t[s[i]-0x20]);
}
printf("\"\n");
```

The result is again "\x5E\x08\x16\x04\x09\x0C\x09\x0E\x05\x25\x5C\x04\x09\x02...".

Applying the decoder routine over the whole executable gives a good view over the containing strings, using that type of encoding:

"*Microsoft Browser Engine*", "*Microsoft Firevall Engine*", "*mdm.exe*", "*ms.mrkva.su*", "*netsh firewall add allowedprogram*", "*MSN Messenger*", "*ENABLE*", "*Download Error.*", "*Download Completed.*", "*Download Started.*", "*Yahoo Spread Started*", "*AIM Spread Started*", "*ICQ Spread*

Started", "*Gtalk Spread Started*", "*Skype spread started.*", "*Pidgin Spread Started*", "*Msn spread started.*" and "*Wrong Params.*"

Quite verbose wich is always good... for me.

5. Bypassing the AV... and Yahoo!'s Updater?!

There are not any serious anti AV protections here except disabling this services
"*YahooAUService*" (YIM's "*Yahoo! Updater*");
"*ekrn*" (ESET's NOD32 antivirus "*ESET Service*");
"*MsMpSvc*" (Microsoft's deprecated "*OneCare Live*" protection system);
"*wuauserv*" ("*Windows Update AutoUpdate Service*").

and terminate and deleting these processes:
"*YahooAUService.exe*" (YIM's "*Yahoo! Updater*");
"*ekrn.exe*" (ESET's NOD32 antivirus "*ESET Service*");
"*egui.exe*" (ESET's NOD32 antivirus GUI);
"*msseces.exe*" (Microsoft's "*Microsoft Security Essentials*")

For some reason Yahoo!'s updater is also considered "harmful" for this malware...

6. Bypassing the Windows Firewall

Bypassing the firewall is made in two methods.

First one is by adding an exception directly to the registry:

A new key named “c:\windows\mdm.exe” is created to "*HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\SharedAccess\Parameters\FirewallPolicy\StandardProfile\AuthorizedApplications\List*", and updated with value "*c:\windows\mdm.exe:*:Enabled:Microsoft Firevall Engine*".
That key is added twice, most probably by mistyping “*HKCU*” to “*HKLM*” for the second reg set.

The second method is by directly using netsh, by executing the command line "*netsh firewall add allowedprogram "c:\windows\mdm.exe" "MSN Messenger" ENABLE*".
This time the program name is changed to "*MSN Messenger*", thus overwriting the previously added key.

7. Run only one instance of the payload

Running more than one instance of a same malware is a bad, bad idea. Things can get ugly very easy , the victim might notice somethings wrong and the developers of this one knew that. So they are using a standard Mutex check to determine if the payload is already running. The name for the Mutex is chosen to be "*Microsoft Browser Engine*".

8. Auto starting

The auto starting with Windows is made by adding a "Microsoft Firevall Engine" registry keys to:
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
HKCU\SOFTWARE\ Microsoft\Windows\CurrentVersion\Run and
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Terminal Server\Install\Software\Microsoft\Windows\CurrentVersion\Run

9. Hiding the files a little deeper

On first run, the malware is located wherever the user put it, so it needs to hide somewhere. Depending on the current user running the malware, the payload will be copied in one of the following destinations : %windir%, "%programfiles%" or... "%public%" (wherever that leads to). Or at least that was the original idea. No matter what user rights you have, the file will always be copied to %windir%.

The destination name is a hardcoded to "mdm.exe" and of course encoded.

Both executables (Loader and Payload) are flagged with SYSTEM, HIDDEN and READONLY flags. Also, the loader will never get deleted.

III. The client-server communication

The most interesting part of course is the client-server communication with the attacker.

- The connection is made over custom TCP protocol.

- The server IP is build depending on the IP of a hardcoded host "*ms.mrkva.su*" by converting it like so:

```
unsigned char ip[] = {198,57,94,27};
int i;
ip[0]=(ip[0]-0x7B);
ip[1]=(ip[1]-0xD2);
ip[2]=(ip[2]-0x49);
ip[3]=(ip[3]-0x11);
for(i = 0; i < 4; i++) {
   if (ip[i]>0x7F) { ip[i]+=0xFF; }
}
printf("%i.%i.%i.%i", ip[0], ip[1], ip[2], ip[3]);
```

At the time of this analysis, four server IP's are available: "*75.102.21.10*", "*74.208.228.202*", "*82.165.11.21*" and "*217.160.92.16*" that are still alive and transmitting packets.

- The port is hardcoded to 5050, by default used by "*mmss*".

1. Client-server communication logic

A standard communication with the attack server looks like this:

step 1. Client connects on the server;
step 2. Server says "*HELLO*";
step 3. Client replays "*HI*";
step 4. Clients replays "*PING*";
step 5. Server says "*DO <.command>*" or "*PONG*";
step 6. Clients replays "*PING*";
step 7...

2. Protocol structure

The connection is separated in two parts: "*Handshake*" ("*HELLO*" -> "*HI*") and "*Conversation*".

The "*Handshake*" starts after connecting on the server, when the client receives a two bytes "*HELLO*", that at the time of writing this material are "*\x80\x40*". Those bytes are used later for XOR encrypt/decrypt the transmitted "Conversation" between the client and the server.

To the server's "HELLO", the client replays with a five bytes of data:

```
struct handshake { // HANDSHAKE with the server
   char xorKey[2] = "\x80\x40"; // The two bytes from the server's HELLO
   byte windowsVersion = 0xXX;  // Where XX may vary from 0x01 to 0x08, or 0xFF
   byte flagFirstRun = 0x00;    // Flag based on the running time
   byte packetCRC = 0xXX;       // Packet checksum
}
```

Depending on the GetVersionExA information, the windowsVersion byte might be one of those:
0x01 = Windows 95
0x02 = Windows NT 4.0
0x03 = Windows 98
0x04 = Windows Me
0x05 = Windows 2000
0x06 = Windows XP
0x07 = Windows Server 2003
0x08 = Windows Vista / Windows Server 2008 / maybe Windows 7 too
0xFF = Unknown windows version

After the "*handshake*", the packets are not plain anymore. They are encrypted instead.

The encryption algorithm looks like:
```
unsigned char p[] = "\x...";  // packet data
int psize = 15;               // size is 15
char key[] = "\x80\x40";      // Key received from the server's "HELLO"
int i, j = 0;
for (i = 0; i < psize; i++) {
   p[i] = (key[j++%2]^p[i])^i;
}
```

At every 30 seconds, the client sends a "*PING*" packet to the server:
```
struct ping { // PING server
   byte packetID = 0x01;      // Packet identification "PING"
   word notUsed1 = 0x0000;    // always set to 0x0000
   char randData[10] = "..."; // Ten random bytes, taken by rand()&0xFF
   word notUsed2 = 0x0000;    // always set to 0x0000
   byte packetCRC = 0xXX ;    // Packet checksum
}
```

Depending on the current server status there might or might not be any attack command available.

If no command is given from the attacker, the server will reply with a "*PONG*" packet:
```
struct pong { // PONG client
   byte packetID = 0x30;               // Packet identification "PONG"
   byte zero = 0x00;                   // Always 0x00
   char unknown1[2] = "\x00\x01";      // ??
   char unknown2[3] = "\x02\xBF\x20";  // Saved but not used
   char unknown3[3] = "\x02\xBF\x20";  // As above
   dword bulk = 0x00000000;            // Four bulk zeros
   byte packetCRC = 0xXX;              // Packet checksum
}
```

The server might use one special packet to force the client to reconnect on the next available server IP:
```
struct reconnect { // RECONNECT client
   byte packetID = 0x0A;      // Packet identification "RECONNECT"
   word zero = 0x0000;        // Not used
   char unknown[11] = "...";  // Unknown but not used
   byte packetCRC = 0xXX;     // Packet checksum
}
```

However, due to misspelling or something, such packet never arrives. However, a packet with packetID = 0x10 comes after every command, that is probably misspelled, because 0x0A in decimal is 10.

The packets might come one by one after every client "*PING*" or they might come as one big chunk of many glued packets.

The standard single "*COMMAND*" reply from the server starts with one or two packets, forming a "*header*" ("*CONFIG*" and "*EXECUTE*" packets), followed by a big (maximum 255) "*command to execute*" data and ends with optional "tail" (usually "*RECONNECT*") packet.

The config packet is used to set the "*echoFlag*" to "*on*" or "*off*" for the command following, and it's structured like so:

```
struct config { // CONFIG server, optional, always before the "EXECUTE" packet
   byte packetID = 0x31;      // Packet identification "CONFIG"
   word zero = 0x0000;        // Not used
   byte echoFlag = 0x0X;      // Might be 0x00 or 0x01
   char bulk[10] = "...";     // Not used
   byte packetCRC = 0xXX;     // Packet checksum
}
```

The execute packet:

```
struct execute { // EXECUTE from the client
   byte packetID = 0x18;      // Packet identification "EXECUTE"
   byte dataSize = 0xXX;      // Size of the command data following this packet
   byte dataCRC = 0xXX;       // CRC of the command data after StringEncoding
   char unknown[11] = "...";  // Not used (seems like)
   byte packetCRC = 0xXX;     // Packet checksum
}
```

The data is standard ASCII string:
".exit" – will terminate the client. That's also the only attack command that doesn't accept any additional parameters;
".m/.me <formatted_message>" - sends over MSN messenger to all of the victims contacts;
".s <formatted_message>" - sends over SkyPe to all of the victims contacts;
".y <formatted_message>" - sends over Yahoo! Messenger to all of the victims contacts;
".g <formatted_message>" - sends over Google's Gtalk to all of the victims contacts;
".i <formatted_message>" - sends over ICQ to all of the victims contacts;
".a <formatted_message>" - sends over AIM to all of the victims contacts;
".p <formatted_message>" - sends over Pidgn/Gaim to all of the victims contacts;
".f <formatted_message>" - sends over Facebook chat to all of the victims ONLINE contacts;
".spr <messenger_id> <formatted_message>" - sends over <messenger_id> from above ("m", "s", "g", etc.);
".dwl <source_url> <destination_path>" - Download file from source_url, saves it at destination_path and execute it;

All of the instant messaging spreading, except the Facebook one are done through FindWindow-Send/PostMessage that actually imitates the normal user behavior, to send a message.
The message is inserted in the message field by first replacing the Clipboards contents, and then simulating CTR+V (paste) command.

For Facebook on the other hand, posting is made by hooking over the users browser (to obtain a valid session), and then parsing the pages to determine all of the online contacts who should get spammed.

The message has sort of formating that may randomize it a bit. Any "#" character that it contains will be replaced with random number from "0" to "9", and every "*" will be replaced with random alphabet from "a" to "z", both lower or uppercase.

An example spam message over facebook will look like this:
"*.f Hey, hows going user:####-****?*"
And the receiver will see it like this:
"*Hey, hows going user:8974-elSz?*"

Since the basic spread commands will only send one IM at a time, ".spr" may send to few of them, with only one attacker packet.
For example, this command will send spam over Facebook, SkyPe and YIM:
"*.spr fsy Hey, hows going user:####-****?*"

The last available attack command is also the most powerful one.
Using ".dwl", the attacker can silently download and execute basically everything – from the latest version of the same malware to a VNC server.

However, no matter what command is send to the client, its encrypted. Twice.
First by applying the reverse algorithm of the string encoding used over the payload's strings (see above), then the result is encrypted with the encryptor used for the packets.

The packetCRC is the sum of all the previous bytes of the packed, ANDed with 0xFF and dataCRC is the sum of the whole data string, AFTER its get string encoded.

Depending on the echoFlag, the client can reply to the server before and after the command is executed with a "*ECHO*" packet, similar to the "*EXECUTE*" one:

```
struct echo { // depending on echoFlag from the "CONFIG" packet
   byte packetID = 0x02;      // Packet identification "ECHO"
   byte dataSize = 0xXX;      // Size of the command data following this packet
   byte dataCRC = 0xXX;       // CRC of the command data after StringEncoding
   char bulk[11] = "...";     // Not used
   byte packetCRC = 0xXX;     // Packet checksum
}
```

Again, after that packet a "*data*" holds the echo message that might be one of those, depending on the command executed:
"*Download Error.*", "*Download Completed.*", "*Download Started.*", "*Yahoo Spread Started*", "*AIM Spread Started*", "*ICQ Spread Started*", "*Gtalk Spread Started*", "*Skype spread started.*", "*Pidgin Spread Started*", "*Msn spread started.*" or "*Wrong Params.*"

IV. S*it no one care about or "i think i forgot to config something..."

1. Procedure that remove a miscellaneous process

There's a procedure that terminate and delete process named "*svhost.exe*". Such file is not Windows native, nor its used anywhere else in the current malware version, so most probably that's the original malware name, and that "terminate and delete" procedure was part of the malware updater method. However, the current sample i got use "*mdm.exe*" as hardcoded file name, so this procedure practically do nothing.

2. List of alternative command hosts

Well, there is one used here! It has 52 entries with alternative command server hosts which are encoded strings, and they are all... decoded with a wrong algorithm, therefore producing empty strings.

Few of the shortest entries in the list looks like this:
\x92\x96\x8D\xCD\xD6\xD2\xCD\x80\x8C\x8E
\x93\x91\x82\xCD\x82\x93\x90\xCD\x8C\x91\x84
\x8E\x82\x90\xCD\x97\x84\x96\x8A\x82\xCD\x80\x8F
\x8E\x82\x90\xCD\x8E\x97\x8A\x8E\x86\xCD\x80\x8C\x8E
I wanted to see what they hold, so I started to look for a weak attack vector of that encoding.

Since every domain name has a "." character somewhere near its end, I made a search for a repeatable character near the string ends:
\x92\x96\x8D\xCD\xD6\xD2\xCD\x80\x8C\x8E
\x93\x91\x82\xCD\x82\x93\x90\xCD\x8C\x91\x84
\x8E\x82\x90\xCD\x97\x84\x96\x8A\x82\xCD\x80\x8F
\x8E\x82\x90\xCD\x8E\x97\x8A\x8E\x86\xCD\x80\x8C\x8E

If "\xCD" is a ".", and the encryption is a simple XOR (because all the strings are with different length, but yet that character is the same in every one of them), then the XOR byte will be (0xCD XOR 0x2E) = 0xE3.

My test deXOR code:

```
char s[] = "\x92\x96\x8D\xCD\xD6\xD2\xCD\x80\x8C\x8E";
int i;
for(i = 0; i < strlen(s); i++) {
   printf("%c",s[i]^0xE3);
}
```

gave me the nice looking string "*qun.51.com*".
Well... bullseye?

The whole host list is decoded as this:
"*pra.aps.org*", "*ols.systemofadown.com*", "*mas.archivum.info*", "*mas.josbank.com*", "*uks.linkedin.com*", "*pru.landmines.org*", "*mas.mtime.com*", "*ale.pakibili.com*", "*beta.neogen.ro*", "*ope.oaklandathletics.com*", "*mas.tguia.cl*", "*old.youku.com*", "*epp.gunmablog.jp*", "*qun.51.com*", "*mas.juegosbakugan.net*", "*mix.price-erotske.in.rs*", "*mmm.bolbalatrust.org*", "*old.longjuyt2tugas.com*", "*xxx.jagdcom.de*", "*mas.ahlamontada.com*", "*mas.0730ip.com*", "*opl.munin.irf.se*", "*mas.univie.ac.at*", "*xxx.stopklatka.pl*", "*ate.lacoctelera.net*",

"*mix.thenaturistclub.com*", "*insidehighered.com*", "*hrm.uh.edu*", "*deirdremccloskey.org*", "*shopstyle.com*", "*www.shearman.com*", "*summer-uni-sw.eesp.ch*", "*journalofaccountancy.com*", "*transnationale.org*", "*unclefed.com*", "*refugee-action.org.uk*", "*southampton.ac.uk*", "*websitetrafficspy.com*", "*stayontime.info*", "*albertoshistory.info*", "*goodreads.com*", "*astro.ic.ac.uk*", "*versatek.com*", "*mcsp.lvengine.com*", "*middleastpost.org*", "*heidegger.x-y.net*", "*erdbeerlounge.de*", "*journals.lww.com*", "*tripadvisor.com*", "*scribbidyscrubs.com*", "*jb.asm.org*", "*screenservice.com*"

I must say here, that none of this hosts are hacked or is known to contain malware, so *System Of A Down* fans, don't worry. ;)

V. Final bits

The malware is simple yet efficient. It has it's flaws and bugs and so on, but still it's efficient. Splitting the functionality to “*spread*” and “*execute*” gives the attacker the simplicity to execute whatever they need to, whenever they wanted, without putting more AV suspicious code into the client's body.

I had some time to code my own server that helped me a lot in reversing the protocol, and also I write a simple “*sniffer*” client that act like the real client and log server's commands. For a security reasons I won't publish their sources.
However, the sniffer was able to catch a command for download and execute from the attack server.
The downloaded file turns out to be a very small and simple Browser “start page” changer, that overwrites IE, FireFox and Chrome's start pages to “*http://zonedirector.com/1/*”.
However, I noticed that it uses the same trick to hide it's API functions, a loader/payload method and self injection like the loader from the client part.
So the attackers are definitely reusing their code, or they might be using a tool for that job.

Another thing that catch my eye was the original name of that downloaded file - “*krava.exe*”.
In my native (Bulgarian) language, “*krava*” means “*cow*” which doesn't make much sense.
However in Latvian and Lithuanian language “*krava*” can be translated as “*load*”, “*cargo*”, etc. so there's a chance that the malware is made in Latvia or Lithuania.