Kindsight

Malware Analysis Report

Botnet: ZeroAccess/Sirefef
February 2012

Kindsight Security Labs

Kevin McNamee
www.kindsight.net

Analysis Summary

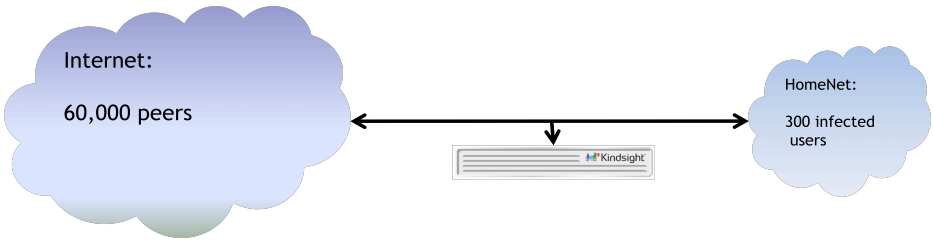
Name: Trojan:Win32/Sirefef.J
MD5: a3c0a2d60e246e5b34f0744939827b5b
Size: 336896 bytes
Source: CleanMX
File Type: PE32 executable for MS Windows (GUI) Intel 80386 32-bit
Sample Collected: 2011-11-04 05:18:17

Introduction

This is an analysis of an encrypted p2p command and control protocol used by malware identified as ZeroAccess, Sirefef, Vobfus and many of other names. The use of this protocol, with slight variations, has been seen in a variety of seemingly unrelated malware samples in our lab, indicating fairly common usage of this component in a number of different malware bundles.

The scale of the infection is quite large. In November we deployed a detection signature in our network sensors and started to see a significant alert level. For example, in one network on a single day, we saw 313 infected computers out of a population of 150K. This indicates that roughly one in every 500 computers is infected with some variant of malware that is using this command and control protocol.

These infected computers were actively communicating with over 60,000 peers on the Internet. One infected computer connected to 5655 different peers in a single day. The average was 972 different peers. It should be noted that to qualify as a “peer” the destination computer must have a live Internet IP address and cannot be located behind a home router using NAT. Since November we have seen p2p connections to over 500,000 different peers. When you consider the NAT issue, this probably means that millions of computers are infected.



Analysis of the encrypted traffic showed that the p2p protocol provides the following commands:

getL – Get a new list of peer IP addresses
getF – Get a file
srv? – Get a list of what files are available

The “getL” command is used to keep in touch with the p2p network. The other commands are used to download and install additional malware components. Unfortunately due to the p2p nature of the botnet, there is no way to trace the files back to an original source and locate the person or group controlling the botnet. However, it may be possible to inject a file into the botnet to affect a takedown.

In addition to the p2p communication, the malware uses an interesting approach to getting past access controls in a Windows 7 environment. It actually downloads a real copy of the Adobe Flash player and starts the installation process. When the user clicks OK for the install to begin and gives it installation permissions, the malware injects its code into the Adobe installation process. The injected thread can now make whatever registry and files system changes it requires to root the system.

In this report we discuss the infection mechanism, show how we broke the encryption algorithm and deciphered the command and control protocol. We then cover the scale of current infection and discuss the various uses of this botnet.

ZeroAccess and Sirefef

The signature that detects this malware identifies it as the ZeroAccess rootkit and many of the AV vendors identify the test samples that trigger this signature as ZeroAccess. However, the first phase of the infection process and the post infection symptoms are not those associated with the previous analysis of ZeroAccess provided by Prevx and a number of other sources.

The second phase of the infection and the “hidden” file system has characteristics that have definitely been attributed to the ZeroAccess Rootkit. It is likely that the malware analyzed here is using ZeroAccess or components of ZeroAccess in phase two if the infection process to mask its presence. The most common name associated with samples that exhibit the p2p behavior and the components that it downloads is Sirefef.

Detailed Analysis

Selecting a Sample

The detection rule that triggers is shown below. The first step was to find a malware sample that would trigger this detection rule.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any
(msg:"Win32.Botnet.ZeroAccess - Runtime Detection";
flow:established,to_server; dsize:20; content:"|E5 AA C0 31|"; depth:4;
content:"|5B 74 08 4D 9B 39 C1|"; distance:5; within:7;
reference:url,www.abuse.ch/?p=3499; reference:url,www.symantec.com/
security_response/writeup.jsp?docid=2011-071314-0410-99&tabid=2;
classtype:botnet; sid:2013911; rev:3; )
```

We have a virus database with over 10M samples and 800K associated pcap files. We regularly run the pcap files against our malware detection rules to determine which samples trigger our detection sensors. In this case we were quite surprised to find that 15,144 different samples triggered this rule.

The sample names, sizes and general characteristics were quite varied, indicating that this particular command & control protocol must be due to a fairly common component in a variety of malware bundles.

Infection

We used a specific sample (VID5248042) for detailed analysis. Additional information on the sample can be found at Virus Total and Threat Expert. Presumably the malware gets downloaded and executed by some form of social engineering, an exploit kit, or by other malware. In the test environment we simply executed the malware file manually. There were two phases to the infection.

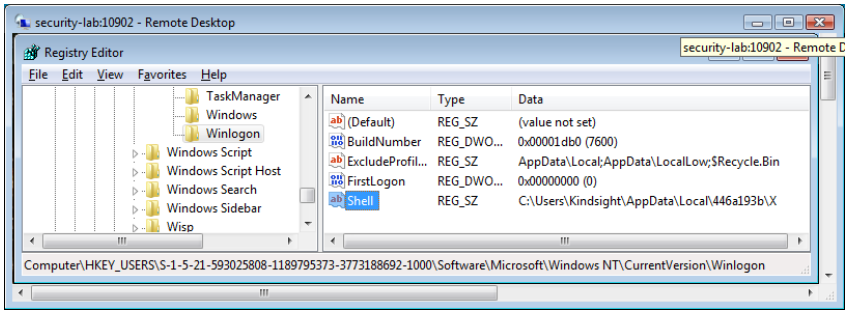
Phase 1

It looks like the first phase of the infection was simply to get a foothold on the victim machine as fast as possible. No obvious attempt was made to hide from detection or permanently establish control.

In the first phase the malware created a subdirectory in C:\users\userid\AppData\Local\446a193b and created files called X and @. The file X, which contains a windows executable, was then run. This injects a copy of the malware into explorer.exe process. It does this by locating the explorer process using ZwProcessOpen(), allocating a block of memory in that process using ZwAllocateVirtualMemory(), copying some shellcode and the packed malware to the process memory using ZwWriteVirtualMemory() and given control via ZwQueueApcThread(). The shellcode, running as part of explorer.exe, then unpacks and relocates the malware executable into

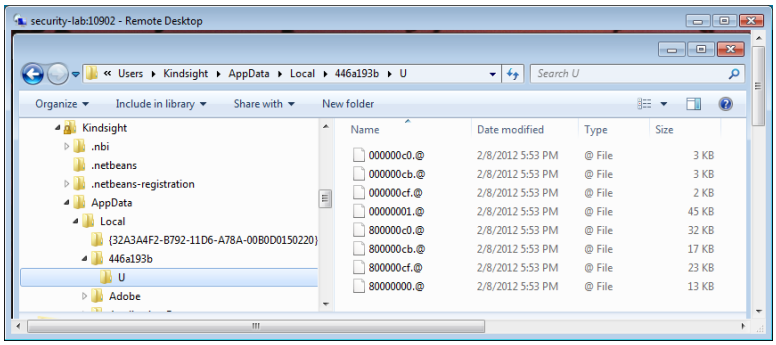
process memory and executes it by calling a function in ntdll. This last part was not traced in detail.

The malware ensures that it will be restarted when the user logs in by the following registry entry.



Registry entry to start the malware

The injected module then opened the file @. This contains a list of 256 peer IP addresses. The malware went through this list attempting to connect to each on TCP port 21810. Once a connection was established, it sends a “getL” command to retrieve the peer’s contact list and updated its own list in the @ file. It continued to do this for all peers that accepted the connection. It reconnected to the first peer to respond and used the “getF” command to download and execute additional malware. These additional files were stored in the U subdirectory. A description of the files is provided in the table below. Periodically, every 20 minutes or so, this process would be repeated.

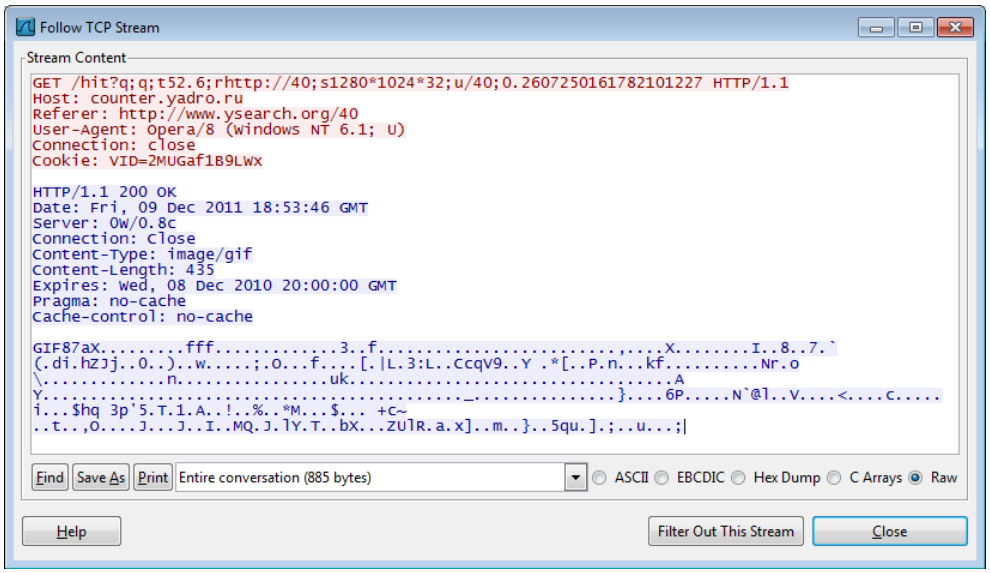


Directory Created

The system then started to exhibited HTTP activity associated with these additional malware files.

Yadro.ru

The malware in the file U\8000000.@ also attached itself to the explorer.exe process. It would wake up every 20 minutes or so and issue a GET request to “counter.yadro.ru”

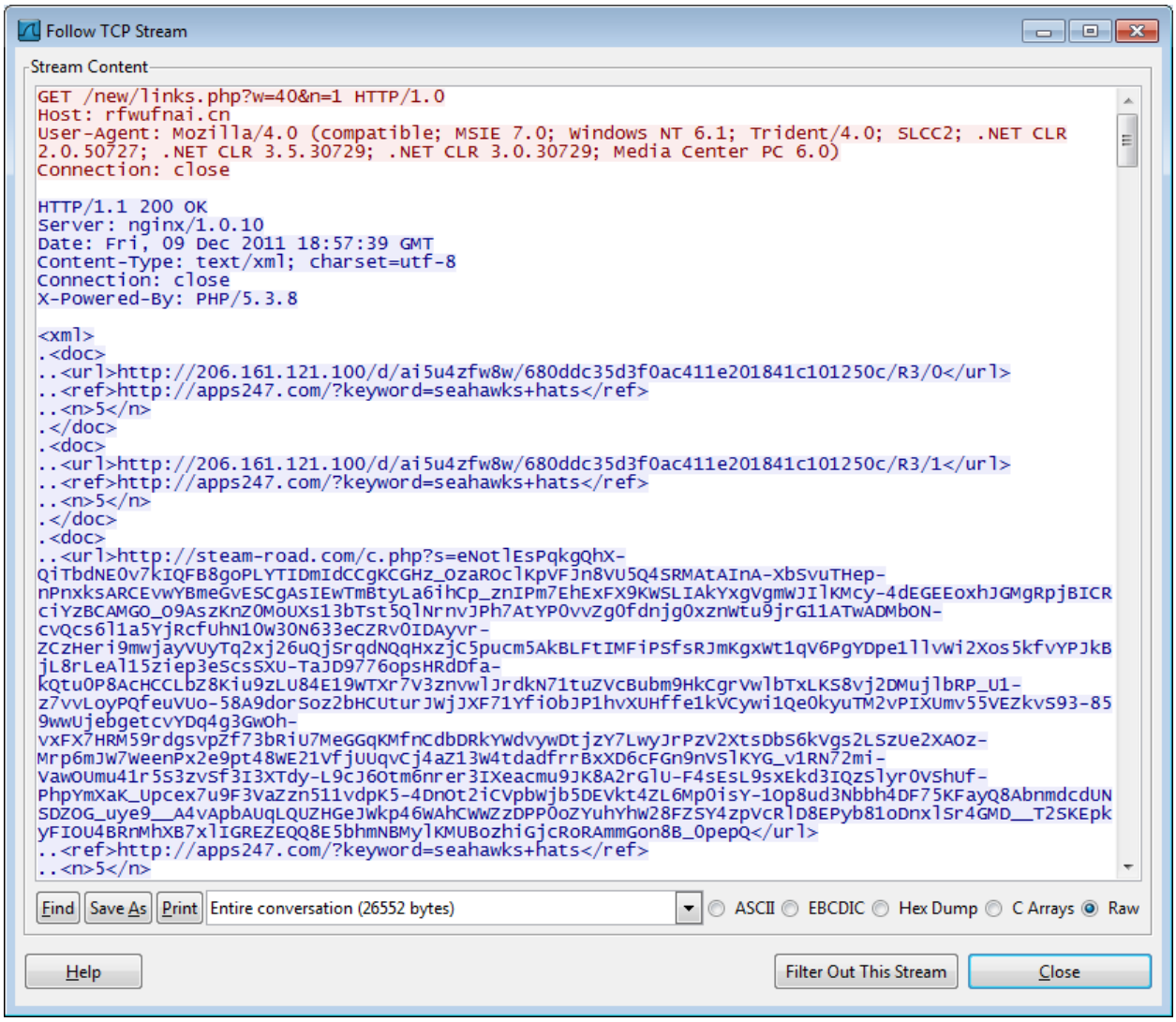


This actually returned the GIF containing a counter as shown below. The thread was not analyzed further. It has debug code built in that can be traced in Ollydbg by activating the “pause on debug string”. It was very nice of them to leave in the debug code.

But it was difficult to see what this is trying to accomplish, other than checking in to indicate a successful infection.

Click Fraud

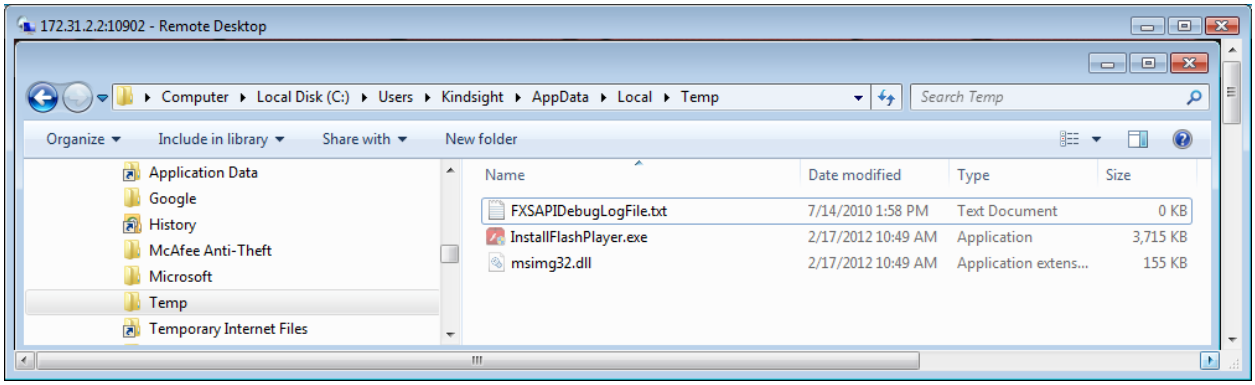
Malware possibly associated with the file U\800000cf.@ created an svchost process. Every 15 minutes or so this process woke up and issued a GET to the URL below.



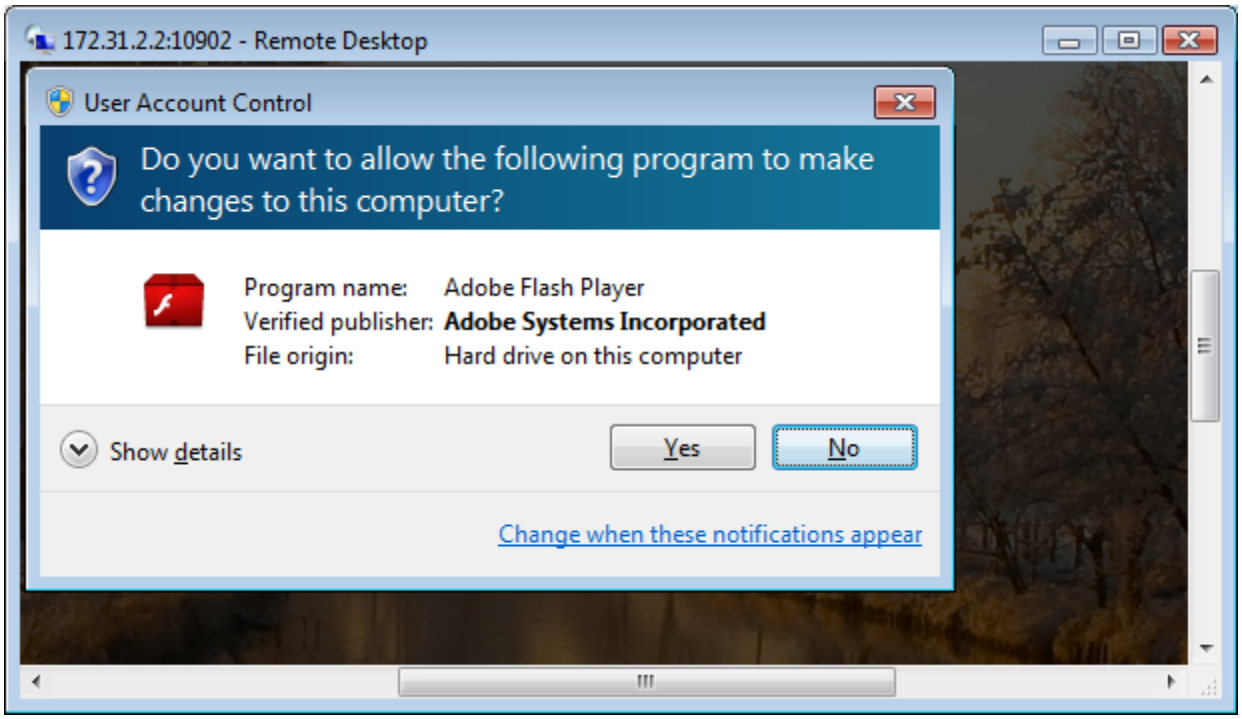
This is obviously a configuration file containing the starting points URLs for an ad-click scam. The process then visits the links in the configuration file using the URLs and referred links provided. The process then appears to crawl the returned pages clicking on various links. Once the latest configuration file is processed (after 10 minutes or so), the process goes to sleep for a while, typically 15 minutes and then issues another “GET /new/links.php” request and repeats the process with a new set of starting points.

Phase 2

The second phase of the infection has definite characteristics that are associated with the ZeroAccess Rootkit. Once the first phase was underway the second phase began by downloading a legitimate copy of the Adobe Flash Player from fpdownload.marcomedia.com. It unpacks the downloaded file and drops it into the temp directory along with an infected copy of msimg32.dll.



It then executed the installation program. This caused the system's UAC to ask for permissions to go ahead with the installation.



Once the permissions are granted the Flash Player installation process is loaded along with the infected DLL file from the same directory. This infected DLL takes advantage of the permissions that it had been granted to create a hidden root-kit environment. This exploit is known to be used by ZeroAccess. See McAfee's Blog post: "ZeroAccess Rootkit Launched by Signed Installers"

1. It creates a hidden partition on the hard drive.
2. It installs a shadow copy of the malware described in phase 1 into this partition. This copy attached itself to the "system process" (pid 4) and uses port 22292. It appeared to run independently from the visible version.
3. It deletes the original executable file.
4. It adds a device service to run a process called 3147163332.exe. This appears to be a watchdog process to protect the malware. It killed our debugger when we tried to attach to it.

If you cancel the Adobe install, the malware simply re-executes it and the permissions dialog box pops up again. At this point, the only way to avoid infection is to kill the malware process or reboot your computer.

Note: This problem was reported to Adobe and has been resolved.

3147163332.exe

On the infected computer we see a process called "991521348:3147163332.exe". It is definitely related to the components described in the phase one infection process because it uses the same hex string to define the service that is used in the directory name used to store the p2p files (X'446a193b'). This process kills the debugger as soon as it tries to attach itself. The following registry entry defines this service.

HKLM\System\ControlSet\service\446a193b\ defines:
ImagePath: \systemroot\991521348:3147163332.exe

The executable is stored in an "Alternate Data Stream" in the file C:\Windows\991521348. This file is visible but has a file size of zero.

AVG's Threat Report for 2011 says that this process is basically a tripwire that allows the ZeroAccess rootkit to detect and kill AV software.

The rootkit creates a “bait” service with a random name… The service is of the structure type SERVICE_OWN_PROCESS and its image file is stored in the alternate data stream (ADS).

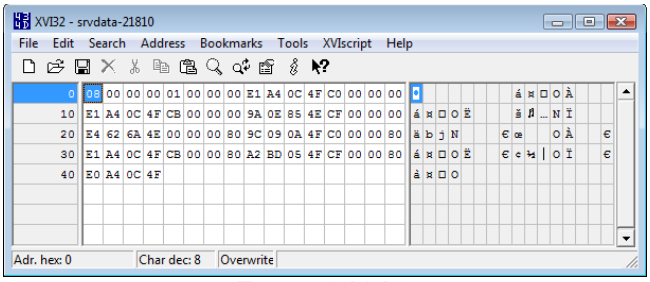
The protection of the process against AV scanners is performed by hooking the nt!IoIsOperationSynchronous function . This kernel-mode function is often called during file open operations. When an application or a driver is caught opening the alternate data stream on that service, the process is killed and the image file’s security descriptor is modified so that it is not allowed to be executed again.

This behavior is consistent with what we observed when we tried to connect a debugger to the process. It’s not clear if it is responsible for any additional behavior.

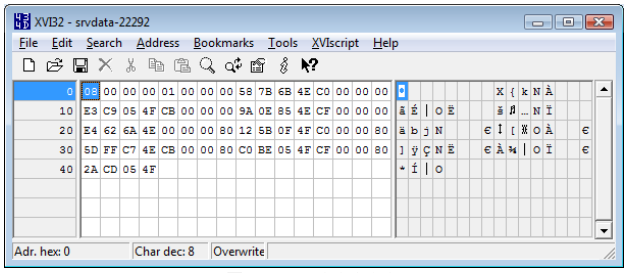
Port 22292

Another instance of the p2p protocol attaches itself to the system process (pid 4). This wakes up periodically and uses port 22292. We were unable to attach the debugger to this process for analysis, so it is difficult to tell if it is the same code using a different port, or a separate implementation. It was also difficult to tell if the same IP addresses were used because they vary with time and are not used in the same order.

Once we got the decryption and file downloads working, it was clear that there are two separate instances of the protocol because the IP address lists are different. This implies that there must be two versions of the “@” file and the U directory, but we could only find one on the system. The screen shots below show the results of the “srv?” command directed at both instances.



From: 21810



From: 22292

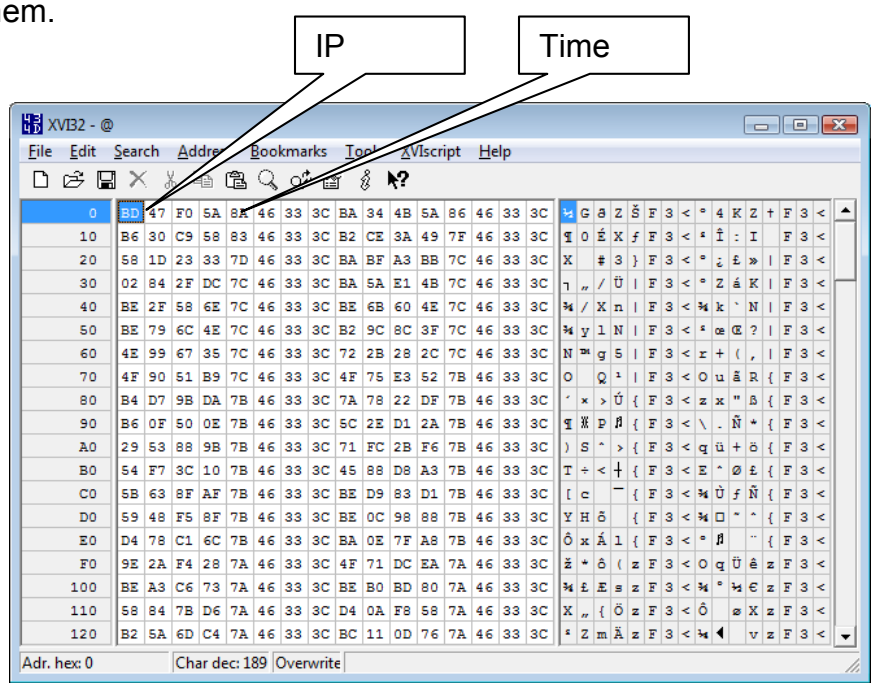
Notice that although the file names are the same the creation times are different. One explanation is that the 22292 version of the malware is using a hidden file system to store its information.

Files Created

The malware created some files in the directory C:\users\userid\AppData\Local\446a193b. The directory name was always the same on the computer under observation, but appears to vary from one infected computer to another. In all cases it had 8 hex digits. The files included:

FileName	Size (K)	Description	Virus Total
X	30	Executable that injects the infection into explorer.exe. This is set as the winlogon shell in the registry.	e09f4a540ae033c49bf740ec732f36a1 Digitala, Inject, Graftor, MaxPlus, Smadow, ZAccess, Sirefef, ZeroAccess
@	2	List of IP addresses of network peers	Nothing
Loader.tlb	3	MIDL type library ?	Nothing
U\800000cb.@	17	Malware executable dll	1e8c4a044453e24e1f26d42a3dc05d79 Sirefef, ZeroAccess, ZAccess, ATRAPS
U\800000cf.@	24	Malware executable dll (does “GET /p/task2.php?w= &i= &n=” with “User-agent: Opera/9” and “Connection: close”)	e46c9ab75e7f2bec02450471b87c8df6 Sirfef, ZeroAccess, ZAccess, Agent.ATAS
U\80000000.@	14	Malware executable dll (looks like yadro.ru module)	52e1c2499d79ae5b2d53a500b181f918 Sirfef, ZeroAccess, ZAccess
The following were downloaded 11 days after the initial infection			
U\800000c0.@	32	Malware executable dll.	273a41f7c65a03f5309defbdf760d71c ZAccess, siggen, SuspectCRC, Sirefef, ZAccess, Cycbot, PMax
U\00000001.@	45	Malware executable dll.	59cc0151f048eff85b5f67824916567e Identified as all of the above plus: Luiha-T, Graftor. SPNR.15A512, Alureon, Rootkit-3105, Smadow, TDSSPack and my favorite Heuristic.BehavesLike.Win32.Suspicious.A
U\000000c0.@	3	Dll with embedded javascript	1cb9d9da501a930f47358712659b7069 Redirector, Artemis, TROJ_AGENTR.CHO
U\000000cb.@	3	Dll with embedded javascript	fe025e8778d66459f3519b8f0199e92c Redirector
U\000000cf.@	2	Dll with embedded javascript	641d61902ae96341113c5c023984b719 Conedex, Downloader, Small, FAKEAV.DAM

The file @ (see hex dump below) seems key to the p2p operation. It contain a list of IP addresses for the peers with a “last updated” time (X’8A46333C’). This file is updated periodically and the infected computer talks with other infected peers. Note that this file is not encrypted. When other peers connect an encrypted copy of the current version of this file is sent to them.



Breaking the Encryption

Traffic Analysis

When the infection occurs the victim sends SYN packets to ports 22292, 21810 or 34354 to a list of preconfigured IP addresses. Different sample choose different ports. One sample started with 21810 and then shifted to 22292. Some malware samples always used the IPs in the same order. Others changed the order. Most of these SYNs return nothing or a RST. When a SYN/ACK is received the victim sends a 20 byte packet and the server responds with 2140 bytes of data. This turns out to be an encrypted version of the file @ that is described above. This is repeated for any servers that respond.

Regardless of the infecting malware or port number used, the initial 20 byte packet sent by the infected computer and the response from the peer showed similar patterns. The parts highlighted are the same for initial contact packets across different malware samples. The un-highlighted parts varied for different sessions.

```
00000000  e5 aa c0 31 48 ba e1 31  31 5b 74 08 4d 9b 39 c1 ...1H..1 1[t.M.9.
00000010  74 c2 a5 f0                                      t...
```

The first part of the response is:

```
00000000  e5 aa c0 31 62 93 7a 5f  31 5b 74 1d 71 93 39 c1 ...1b.z_ 1[t.q.9.
00000010  eb 7c b6 83 a0 60 ac 8d  04 28 5c 11 35 96 94 d8 .|...`.. .(\.5...
```

Again the highlighted parts are the same across all sample observed. Also there is an interesting repeating pattern in the rest of the response. This is shown in the large hex dump below. Regardless of which server responded the highlighted bytes always match. This seems to indicate that the information is not encrypted using any normal cipher. It is hard to see how the pattern of 2-bytes the same, 6-bytes variable could be the result of encryption. There is also a large section at the end of the response that is always the same, regardless of the server that responded.

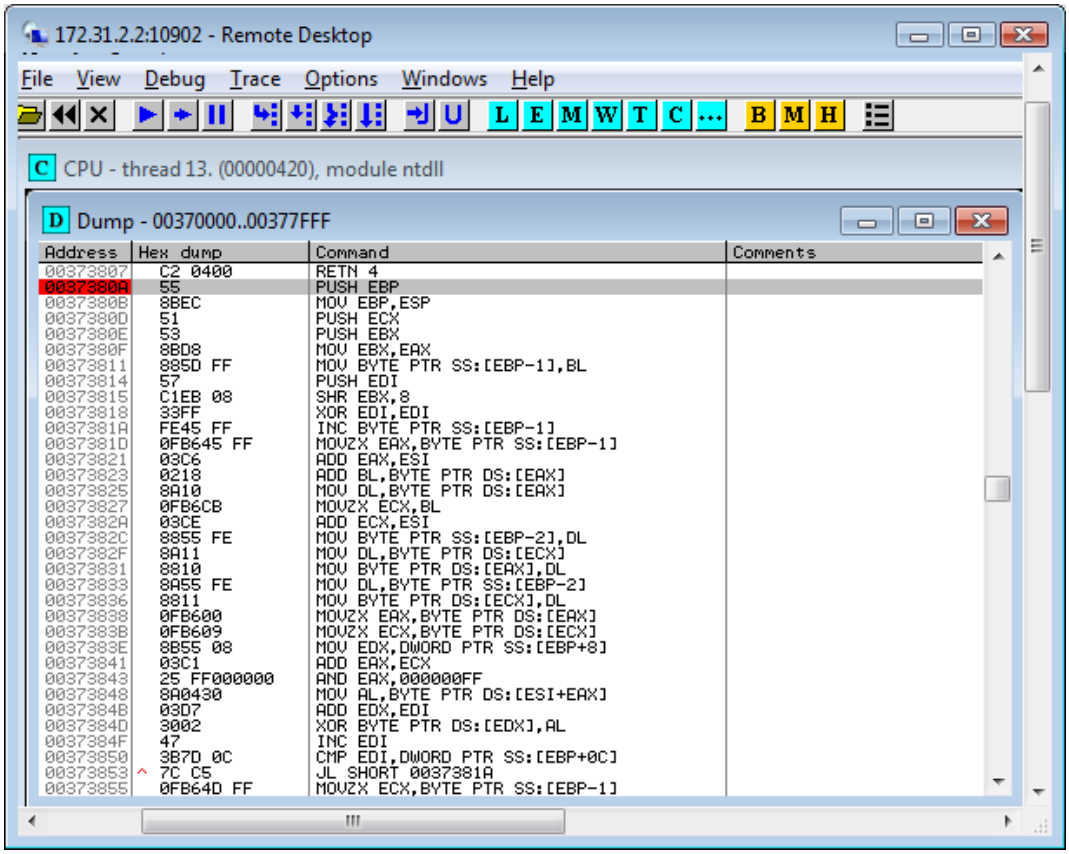
```
00000000  e5 aa c0 31 48 ba e1 31  31 5b 74 08 4d 9b 39 c1 ...1H..1 1[t.M.9.
00000010  74 c2 a5 f0                                      t...
    00000000  e5 aa c0 31 62 93 7a 5f  31 5b 74 1d 71 93 39 c1 ...1b.z_ 1[t.q.9.
    00000010  eb 7c b6 83 a0 60 ac 8d  04 28 5c 11 35 96 94 d8 .|...`.. .(\.5...
    00000020  8a c3 86 4c cd 5f 9d bf  a1 37 41 6e 7c 8f de 30 ...L._.. .7An|..0
    00000030  16 dc 54 1d 8a 61 6b 4b  34 e3 1d df 14 99 9d 0c ..T..akK 4.......
    00000040  5c 2d 94 08 71 00 d6 db  f0 b3 59 96 da e4 d9 93 \-..q... ..Y.....
    00000050  40 5f 54 0b fd 2c a5 78  dc cb f6 c5 5c 75 27 e4 @_T..,.x ....\u'.
    00000060  20 e2 55 69 12 6a db 43  85 12 c0 48 4f 57 ee 33  .Ui.j.C ...HOW.3
    00000070  67 83 99 00 42 cd 68 5a  d9 f9 e8 48 3b 70 e0 db g...B.hZ ...H;p..
    00000080  84 a0 62 7b 5c 2d d6 e1  96 1b 85 cb b5 05 bd 81 ..b{\-.. ........
    00000090  60 12 34 33 5f 15 3f 21  a4 7c 6f 73 06 16 0f 72 `.43_.?! .|os...r
    000000A0  a9 aa 0c ea 96 b0 3d cf  7a 80 d1 1f 40 b8 78 69 ......=. z...@.x
    000000B0  b0 d2 2b cf b9 6f 1d 46  6e ab 0a b4 71 70 c7 c9 ..+..o.F n...qp..
    000000C0  f1 42 19 a9 f0 a3 9f 57  e8 52 1e 43 26 45 c4 a4 .B.....W .R.C&E..
    000000D0  fd 24 53 be ea 84 ba 98  ae fe 96 40 5e a0 47 e8 .$S..... ...@^.G.
    000000E0  6b 02 95 1c da 4a 6f f9  79 f0 b1 10 6c 91 da cf k....Jo. y...l...
    000000F0  cf 45 44 bf da 01 82 1b  a4 c3 13 69 73 90 2f 9b .ED..... ...is./.
    00000100  78 0e 28 bf 74 37 92 6e  70 3e 6d 95 14 a6 ed 81 x.(.t7.n p>m.....
    00000110  c1 4f 3e b9 97 13 33 4b  2c 0b 78 13 14 c8 99 f0 .O>...3K ,.x.....
    00000120  8a fa da b8 3c 5b e6 39  38 96 41 b3 c3 16 f8 83 ....<[.9 8.A.....
    00000130  cc 92 bd a4 da de f2 24  37 1a a9 2b 65 28 67 41 .......$ 7..+e(gA
    00000140  b9 16 32 5c 20 43 5c dc  47 16 19 13 f9 5c 10 fa ..2\ C\. G....\..
    00000150  b1 a1 87 0b e1 a3 82 f8  b9 c9 23 02 11 67 05 37 ........ ..#..g.7
```

As it turned out these two byte fields were usually zeros in the clear text version and always encrypted to the same values.

Another interesting observation was that often the first exchange would include an out of order ACK from the server side. This could have been caused by packet loss on the network, but it happened with different servers, so it may be related to some bug in the server code.

Encryption Algorithm

We ran the malware under a debugger to figure out what kind of encryption this could be and decode the command and control protocol. The encryption/decryption code is shown below:



We reverse engineered this algorithm. The C-code is shown in the box on the following page. In the malware, a 256 byte array is created using the MD5 of the key (X'FE4367CD'). This array is hardcode in the C-code as the "bytes_orig" array. The encryption/decryption loop then uses a strange substitution algorithm to generate a bytes stream that is XORed with the input to encrypt or decrypt the data.

The encryption routine is at location X'380A' in the module. The sequence of events leading to encryption starts at X'308A' and sort of looks like in pseudo code:

```
InitBuffer()          /* move in X'FE3467CD' and appropriate data and zero crc field */
computeCrc32()        /* compute crc */
InitByteArray()       /* Uses the md5 of the secret key  to build 256 byte array */
crypt(buf, len)       /* this generates the XOR byte stream using array from above */
```



```
void crypt(char * in, char * out, int len)
{

  // The following implements the encryption/decryption function for ZeroAccess.
  // I'm not sure what encryption algorithm this is. The table below is used to generate a stream of
  // bytes that are used to XOR the input. This byte stream looks pretty random, but is the same
  // each time. The table was generated from an md5 hash of the "key" that is the first 4 bytes of
  // each transmission. I did not implement the table generation algorithm, but just copied it from
  // running malware sample.
  //
  // The algorithm implemented below was reverse engineered from the malware. On each iteration
  // it grabs a couple of bytes from the table and swaps them in the table. It then adds them
  // together and uses that as an index to retrieve the value to be XORed from the table.

  int i;
  unsigned char bytes_orig[256] = {
        0x07, 0x71, 0x3E, 0x01, 0x92, 0x77, 0x67, 0xE2, 0x6E, 0x55, 0xC2, 0xF7, 0x12, 0x82, 0x8D, 0xE6,
        0x32, 0x96, 0x15, 0x2A, 0x5B, 0xAE, 0x24, 0xD6, 0x60, 0x02, 0xE9, 0x2E, 0x59, 0xA1, 0x7B, 0xF6,
        0xD7, 0xA7, 0x5E, 0xD3, 0x28, 0x1B, 0xC9, 0x1D, 0x0D, 0x86, 0xB5, 0x68, 0xA3, 0x31, 0x7D, 0x5D,
        0xA5, 0xED, 0x5F, 0x4D, 0x76, 0xCC, 0xAD, 0x7A, 0x3D, 0x33, 0xA4, 0x4F, 0x2C, 0x1E, 0x21, 0x97,
        0xDE, 0xC7, 0x3F, 0xCB, 0xF9, 0xB2, 0xBE, 0x3B, 0x9E, 0xA0, 0xEF, 0xA2, 0x29, 0x1C, 0x87, 0x41,
        0xB8, 0x84, 0xCF, 0xE7, 0x95, 0xF5, 0xC0, 0x03, 0x22, 0x9D, 0x0F, 0x98, 0xF4, 0x85, 0x4A, 0x09,
        0x99, 0x63, 0x46, 0xD0, 0x89, 0xF2, 0x26, 0xEC, 0x8A, 0x6D, 0xCE, 0x50, 0x61, 0xAB, 0xAA, 0xBB,
        0x0C, 0xB4, 0xBF, 0xC5, 0x40, 0xAF, 0x3C, 0x8C, 0xC4, 0x39, 0xDF, 0xD5, 0xA9, 0x51, 0xD4, 0x8B,
        0xC1, 0x04, 0x80, 0x91, 0x7E, 0x93, 0x0E, 0xD8, 0xE8, 0x90, 0xCA, 0xD2, 0x9B, 0xF3, 0x45, 0x4C,
        0x9A, 0x43, 0x62, 0xF1, 0x5A, 0x8E, 0x9F, 0x49, 0x48, 0xB1, 0x2F, 0x23, 0x79, 0x37, 0x18, 0xFF,
        0xA6, 0xFA, 0x25, 0x4B, 0xEA, 0xE3, 0xB6, 0x6C, 0xE0, 0xF8, 0x78, 0xB9, 0x6F, 0xC8, 0x6B, 0x0A,
        0x20, 0xDA, 0xEB, 0x81, 0x2B, 0x2D, 0xDB, 0x4E, 0x1F, 0x27, 0x36, 0x42, 0x8F, 0x06, 0xFD, 0x65,
        0x58, 0x9C, 0xD1, 0xB7, 0x47, 0xD9, 0xFB, 0x88, 0x14, 0x35, 0xDD, 0x11, 0x6A, 0xBC, 0x72, 0xAC,
        0xC6, 0xFC, 0x30, 0x08, 0x64, 0x66, 0x44, 0x70, 0xBA, 0xE5, 0x57, 0xC3, 0xB0, 0x5C, 0x17, 0xA8,
        0x56, 0x3A, 0xFE, 0x94, 0x54, 0xB3, 0x7F, 0x52, 0x10, 0x69, 0x73, 0x05, 0x16, 0x1A, 0x74, 0x75,
        0x83, 0x0B, 0xE1, 0x38, 0x34, 0xF0, 0xCD, 0x19, 0x13, 0x7C, 0xDC, 0x00, 0x53, 0xE4, 0xBD, 0xEE
 };

  unsigned char bytes[256];

  // initialize the byte array
  memcpy(bytes,bytes_orig,256);

  // initialize the loop counters and indexes
  unsigned char bi1 = 0; // byte index 1
  unsigned char bi2 = 0; // byte index 2
  unsigned char b1  = 0; // byte value 1
  unsigned char b2  = 0; // byte value 2
  unsigned char bi  = 0; // index to xor byte
  unsigned char xor_byte = 0;

  // loop to encrypt/decrypt
  for (i=0;i<len;i++)
  {
    bi1++;
    b1 = bytes[bi1];
    bi2 = bi2 + b1;
    b2 = bytes[bi2];
    bytes[bi1] = b2;
    bytes[bi2] = b1;
    bi = b1+b2;
    xor_byte = bytes[bi];
    out[i] = in[i] ^ xor_byte;
  }
}
```

Encryption Algorithm in C

Command & Control

With the encryption algorithm figured out, the command and control protocol turned out to be pretty straight forward, except for the weird encoding used for the commands.

Packet format

The basic packet format for the C&C protocol as follows:

X'FE4367CD'	CRC	Cmd	Length	Data

CRC - A crc of the entire buffer calculated with this field set to zero
CMD - This is the command field
Length - Length of the data field in bytes
Data - Data depends on what command is used.

The CMD field is an ASCII character string that is encoded as a little-endian integer value. The following valued are checked for in the code:

X'46746567' - "getF" (gets a file)
X'4C746567' - "getL" (gets a list of peer IP addresses)
X'7372763F' - "srv?" (gets a list of files for download)

The command decode routine is at location X'2F59' in the module. It actually uses integer value subtractions to match the strings, which are stored in little endian order. Bizarre!

```
SUB   EDX,67657446          /* is it getF */
JE    30AB
SUB   EDX,6                 /* how about getL */
JE    2FD7
SUB   EDX,0C0D01F3          /* how about srv? */
JNE   30A0
```

The entire data stream is then encrypted using the XOR algorithm described above.

getL

This asks the peer to send its latest list of peer IP addresses. Peers update this periodically by contacting other peers and use the file called "@" to maintain an up to date version of this information. A 4 byte time stamp appears to be used to track how current the information is. The request looks like:

X'FE4367CD'	CRC	X'4C746567'	X'04000000'	X'6CC5665C'

The command is X'4C746567' which in ascii is "Lteg" (which reversed is "getL"). The length is 4. The data is X'6CC5665C' (I don't know what this is for).

The peer responds with a similar header with the command field set to “setL” and the length set to the size of the data field. This is then followed by 256 IP address entries in the format shown below:

Num	IP1	T1	. . .	IPn	Tn

Here “num” is the number of IP entries. These are stored as pairs, containing an IP address and the time in seconds since the IP1 address was last confirmed. The time on the wire is send as seconds since the last update occurred. The time in the “@” file appears to be in seconds since 1980?

getF

This is used to retrieve files from the peer. The files are from the directory C:\users\userid\AppData\Local\446a193b as described earlier in the document. The command to retrieve the file is of the form:

X’FE4367CD’	CRC	X’46746567’	X’04000000’	X’aaaaaaaa’

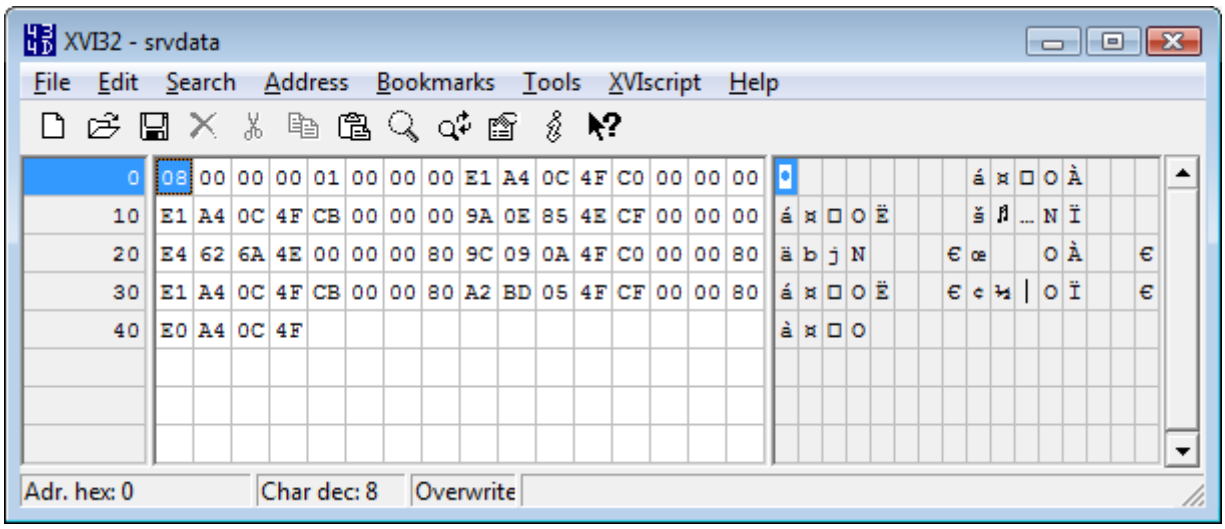
The command is X’46746567’ which in ascii is “Fteg” (which reversed is “getf”). The length is 4. The data field is 4 bytes long and is converted to a hexadecimal file name with “.@” appended to the end of it. For example if the field contains X’CF000080’, the file name will be 800000cf.@ (don’t forget, it’s a little endian machine).
The peer responds with a similar header with the command set to “setF” and the length field set to the file size. The data contains the encrypted version of the file.

srv?

This command asks the peer to send a list of the files it has available for download. The command is as follows:

X’FE4367CD’	CRC	X’3F767273’	X’04000000’	X’00000000’

The command is X’3F767273’ which in ascii is “?vrs” (which reversed is “srv?”). I’m not sure if its required, but I set the length to 4 and put zeros in the data field. The peer responds with a matching header followed by a list of filenames coded in hex as indicated in the “getF” command. The data part of the response is shown below. The first word is the number of entries (in this case 8). This is followed by file name entries that consist of the 4 byte hex filename and a 4 byte number that may represent how old the file is.



The response above corresponds to the list of files in the U subdirectory. The table below shows the list of names, the extra word value and the file creation date from the peer system.

File name	Word Value	Creation Date
00000001.@	4F0CA4E1	Jan 10th 2012
000000c0.@	4F0CA4E1	Jan 10th 2012
000000cb.@	4E850E9A	Sep 29th 2011
000000cf.@	4E6A62E4	Sep 9th 2011
80000000.@	4F0A099C	Jan 8th 2012
800000c0.@	4F0CA4E1	Jan 10th 2012
800000cb.@	4F05BDA2	Jan 5th 2012
800000cf.@	4F0CA4E0	Jan 10th 2012

Clearly the “word value” associated with the file name, must be the creation date.

We tried dropping file into the U directory to see if the peer would serve them up as part of the list. This was not successful, so the malware may maintain a list separately from the directory, or perhaps has some mechanism to verify the authenticity of the files. We created a text file called 000000cc.@ and copied 00000001.@ to 00000002.@. Neither of these new file appeared on the list. I even tried restarting the computer, all to no avail. However, these files were delivered when the “getF” command was used.

Further analysis of the code showed that the malware only shows files if the milliseconds field in the time_field structure for the file is zero. This is a bit weird, but perhaps this check is to make sure it doesn't accidentally start serving files just because someone dropped them into the U directory. The probability of the milliseconds field being zero is 1000 to 1.

Mapping the Botnet

As mentioned in the introduction, in a live network on a single day we saw 313 infected computers actively communicating with over 60,000 peers on the Internet. With the encryption broken and the C&C decoded, we could now map the size of the botnet at a given time.

We wrote a mapping program that started out using the list of 256 IP addresses from our infected host. It then contacted those IP addresses and asked them for their lists. The program visited any new IP addresses in a recursive loop, keeping track of each IP address visited. After 24 hours it found 121,000 infected hosts.

Since December there have been over 500,000 connections to unique peers observed by our network sensors.

Botnet's Purpose and Operation

Given the limited C&C described above, one can speculate as to the purpose of the botnet and how it is operated. The files dropped into the U directory are definitely malware, so the most likely purpose of the botnet is to download and execute additional malware files.

The bot maintains an up to date list of peers using the "getL" command on a fairly frequent basis. This provides it with at least 256 active peers. Less frequently it will issue a "srv?" command to selected peers. This will provide it with a list of any new files that are out there.

It used the "getF" command to download and install the new files. We have not observed the bot for long enough to determine the relative frequencies of these operations. The bot under observation has downloaded files at least 3 times in the past month.

The bot operator need not have a special process for distributing new malware files. All he has to do is drop a fresh file (with the millisecond field set to zero) onto a peer and the file will be automatically distributed to the botnet.

Possible Take Down

One way to take down this botnet is to inject a poisoned pill into the U directory of one of the peers. The poisoned pill would be an executable that kills the infected explorer process and deletes the infected directories. This has not been tested as of yet, but it is fairly straightforward to test.

Another possibility is to take it down by interfering with the IP address exchange. Injecting false peer addresses into the "getL" response could disable a large number of peers.

