SOPHOS

simple + secure

What is Zeus?

By James Wyke, Threat Researcher, SophosLabs UK

A SophosLabs technical paper - May 2011

Contents

Abstract

Zeus or Zbot is one of the most notorious and widely-spread information stealing Trojans in existence. Zeus is primarily targeted at financial data theft; its effectiveness has lead to the loss of millions worldwide. The spectrum of those impacted by Zbot infections ranges from individuals who have had their banking details compromised, to large public order departments of prominent western governments.

We will explore the various components of the Zeus kit from the Builder through to the configuration file; examine in detail the functionality and behaviour of the Zbot binary; and assess emerging and future trends in the Zeus world.

Introduction

Zeus (also known as Zbot) is the name of a toolkit used to create a particular strain of information stealing Trojans. The bots created by the kit run silently in the background on compromised computers, harvesting information and sending it back to the bot herder. The main focus is to steal online banking details and other login credentials but the range of different types of data theft is extremely broad.

The kit is obtained on underground forums with older versions available for free and the newest, fully-featured versions costing several thousand dollars.

The Zeus kit is very simple to use, requiring little technical knowledge. As a result, huge numbers of independent Zeus-created botnets exist, all with their own controllers. A by product of this is that we in the AV industry see huge numbers of Zbot samples that seem to bear no relation to each other, as each botnet owner packs and obfuscates their samples in different ways. Some of these self-contained botnets have been hugely successful with stories in the press of some operations managing to steal hundreds of millions of dollars.

Brief History of Zeus

Although the overall aim of Zbot (that of information theft) has remained the same since it first emerged, there have been several noticeable changes along the way in how it achieves that goal.

The earliest versions were notable by the consistent filename that the bot executable was given when first run on the target system and the filenames given to the data files.

Early samples would use the following filenames for the executables:

ntos.exe, oembios.exe, twext.exe

Data files were stored in the following directories:

<System>\wsnpoem

<System>\sysproc64

<System>\twain_32

The next major version predominantly used "sdra64.exe" for the executable name and stored the data files in "<System>\lowsec".

The most recent version of Zbot stores the executable file with a random filename in a newly created, randomly named folder in the Application Data folder of the executing user.

Components of Zeus

The Builder

Each prospective Zeus botnet owner must create their own bot executables that they will distribute to their victims. To do this the Zeus kit includes a builder (Fig 1).

Each customer uses the builder to create both the encrypted configuration file and the bot executable that is specific to the customer. The executable will be unique for each customer (even if two customers use exactly the same version of the builder) due to the configuration file URL and the key needed to decrypt the configuration file that are embedded into the executable.

First the configuration file needs to be built which includes all the essential information that makes the bot do anything useful (more in the next section), including the URL where this file will be located. The file must first be edited with the bot owner's settings and then built using the "Build config" button. The builder will then convert the text file into the binary format expected by the executable, compress and encrypt it. The botnet owner then places the encrypted file at the URL they specified during the build, to be retrieved by the bot upon execution (Fig 2).

Fig 1. Zeus Builder

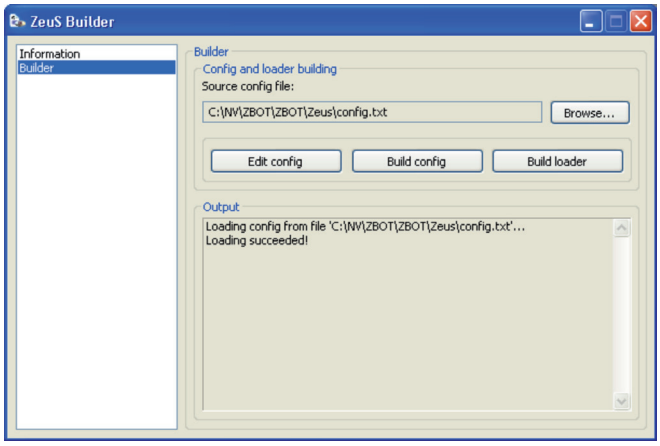
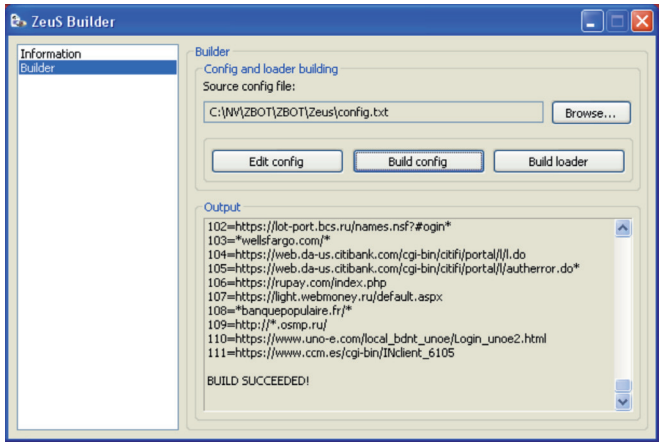


Fig 2. Zeus Build config



Then the executable can be built with the "Build loader" button. The Builder will embed the information needed to retrieve and decrypt the configuration file into the Zbot binary before packing it with its own custom run-time packer. Generally speaking, Zeus customers will then pack the executable again with some other packer (Fig 3).

The Configuration File

The configuration file component of the Zeus kit is absolutely essential if the bot is to do anything useful. It is a separate entity to the executable, downloaded during execution.

This file contains (amongst many other things) the address to which all the stolen data is sent. If the configuration file cannot be retrieved then the bot will not know where to send its stolen data.

The format the file takes is a series of blocks that enable and customise the various functionalities that Zbot offers. The following screenshot shows the file before it is packaged by the builder, as you can see from the top of the file this is version 1.2.17.19 (quite old, we have now seen beyond version 2.1)(Fig 4).

It is divided up into sections that start "entry" with the two main being "entry 'StaticConfig'" and "entry 'DynamicConfig'". These two sections deal with the settings that will be hardcoded into the binary and the settings that will be written into the configuration file and downloaded at runtime.

The static options include timing options (how long to wait between attempting to download the config file etc), the URL from which the configuration file is downloaded, and a URL that is used to check the external IP address that the bot is phoning home from. These will be written into the binary when it is distributed.

Fig 3. Build loader

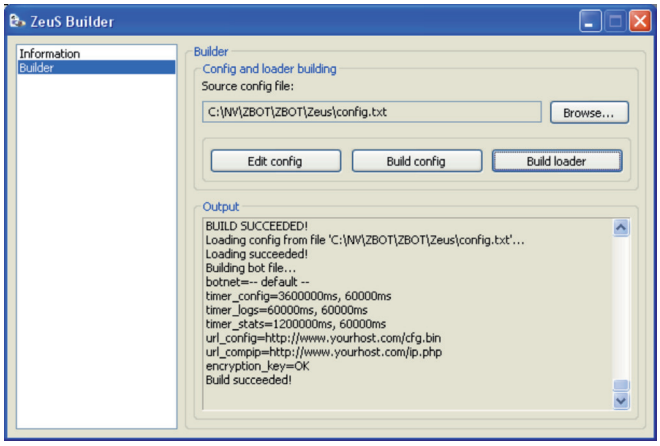
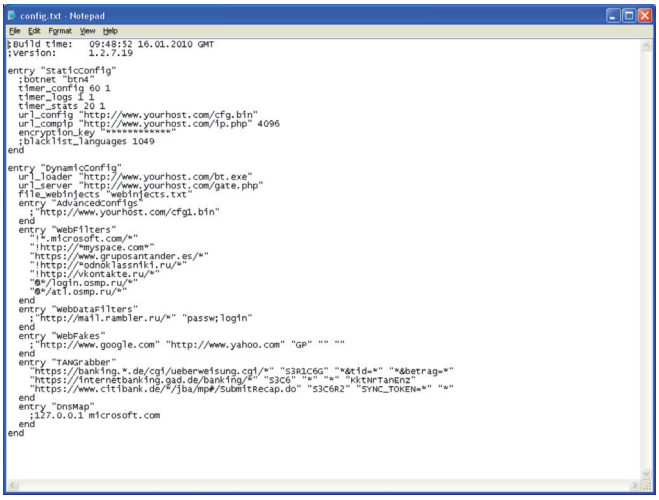


Fig 4. Configuration file



The dynamic options mainly centre on what particular web addresses the bot owner wants to target but there are also several housekeeping entries, including:

- a URL from which a new Zbot executable will be downloaded
- a URL to which stolen data is sent back
- a URL at which a further configuration file can be downloaded

The other dynamic options include:

- A set of URL masks that enable or disable logging for those URLS
- A set of URL pairs where one URL is redirected to the other URL
- A group of URL's from which TAN's (Transaction Authentication Number) will be harvested
- A set of IP/domain pairs that will be written into the hosts file to hijack DNS requests.
- A set of URL masks each with a corresponding block of HTML that will be injected into any page whose request matches the URL mask (WebInjects)

The last item is where the Zeus bot owner can really capitalise financially on their installation. The owner can inject any data they wish into any webpage such as extra fields in online banking web pages that ask for ATM pin numbers and social security numbers. The following screenshot shows a typical item from the WebInjects section (Fig 5).

Fig 5. WebInjects section

```
set_url https://www.wellsfargo.com/* G
data_before
<span class="mozcloak"><input type="password"*</span>
data_end
data_inject
<br><strong><label for="atmpin">ATM PIN</label>:</strong> <br />
<span class="mozcloak"><input type="password" accesskey="A" id="atmpin" name="USpass"
data_end
data_after
data_end
```

This shows that the code snippet will be injected into any URL that contains "https://www.wellsfargo.com", where the code will be injected (after the data in "data_before"), and the code itself which is extra fields in the form requesting "ATM PIN" etc.

All the dynamic configuration data goes into the configuration file that is stored on the server. The bot will then periodically query the URL for the file and process the data it contains. In this way the bot owner can easily change the behaviour of the bot by uploading a new configuration file to the server.

The Exe File

The exe file that is built by the builder component is to be deployed by the botnet owner. Different Zeus kit customers using the same version of the kit will produce almost exactly the same exe file, with the most important difference being the location of the configuration file which gets embedded into the binary by the builder. This is essentially the only thing that differentiates one Zeus kit created botnet with another – the configuration details. The functionality and the behaviour will always be the same.

The Server

The server component of the Zeus kit is a collection of php scripts that allow the owner to monitor the status of their bots, issue commands to them and retrieve the information that they have collected.

The interface is very user-friendly and a vast array of information about the botnet is available to the owner.

Functionality of the Zbot Binary

Execution Overview

In very general terms Zbot performs the following actions:

- Copy itself to another location, execute the copy, delete the original
- Lower browser security settings by changing IE registry entries
- Injects code into other processes, main process exits
- Injected code hooks apis in each process
- Steals several different type of credential found on the system
- Downloads config file and processes it
- Uses api hooks to steal data
- Sends data back to C&C

The last two major versions of Zbot have several distinct differences in their execution.

The previous version copied (I use term "copy" loosely here as changes are made to the binary which I will come back to later) itself into the system directory, usually using a name along the lines of "sdra64.exe", created a folder, also in the system directory, typically called "lowsec" that contained the downloaded configuration file and temporary file holding stolen data while it was waiting to be sent to the C&C server. This version added an entry to the "Userinit" value of the "HKLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon" registry key, in order to run on system startup. This version also hooked several ntdll.dll apis including NtQueryDirectoryFile in order to hide its files. This usermode rootkit is not present on the more recent version.

The version I will concentrate on most (version 2), copies (again loosely used) itself to the user's "Application Data" folder using a randomly generated filename and directory name. The temporary data file is also stored under %AppData%, again with random names, and the configuration data is downloaded into the registry rather than onto the disk. This version creates a runkey under HKCU. The use of Application Data and HKCU over the system directory and the Winlogon key means that many different Zbot botnets can infect the same machine, and different users can be individually infected.

Anti-Checksum Based Detection

For quite some time Zbot has used techniques to make full file checksum based detection less than effective. Pre-version two a variable amount of random data was appended to the file when it was copied into the system directory (hence loose use of the term "copy").

Version two uses a more complicated technique to ensure that the dropped file will not have the same checksum as the original dropper.

When the version two executable "copies" itself to the user's "Application Data" directory, a small (just under 0x200 bytes) block of encrypted data is embedded into it. This block contains (amongst other things) the path that the file has been dropped to and a GUID generated from the disk that it is on. Because the pathname is randomly generated this block will be different each time the file is executed, thus making it extremely unlikely that two identical files will be produced.

Detailed Run-Through of Zbot Execution

For this detailed run-through I will concentrate on the more recent version of Zbot (2 and later). Specifically the sample I'm using (sha1: 014e733640898f169e61074dae-f35e2f14267bbb) gives its version as "02.00.06.05".

Unpacking

Although I won't go into the details of the various packers used by Zbot, it is worth mentioning that the file will almost always be packed. The builder itself will pack the file with its own custom, packer, but it is rare that a Zbot customer will not pack the file again with some other packer.

It Begins...

Initially, the sample will resolve and store some imports from ntdll.dll, retrieves some general information about itself – PID, OS version, whether it's running under WOW 64, process access level; then it will check its command line arguments. The sample will run just fine without any arguments but there are a few that can be supplied, the most interesting of which is probably "-i" which will cause the sample to display a message box giving information about the sample (including version) and terminate (Fig 6).

Dropper or Droppee?

The Zbot file will then establish whether it is the dropper (in which case it needs to copy itself to %AppData%) or if it has already been dropped (in which case it needs to inject itself into other processes).

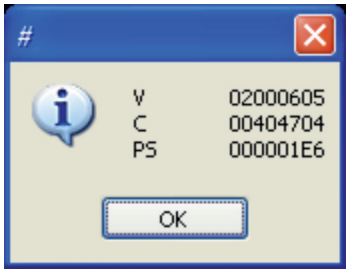
First, the sample reads its own executable from the disk into dynamic memory. It then finds the address of the start of the first section named ".data" and copies 0x200 bytes from that location to the heap. These bytes are then decrypted using RC4 and the final dword from the decrypted RC4 block is used as a flag to indicate whether we are the dropper or droppee.

To begin with we will assume we are the dropper.

Decrypt the Dropping Routine

A mutex is then created with a cryptographically derived name, designed to be unique so that only this bot sample will create a mutex with that name (this is a common pattern throughout Zbot's execution). This ensures that only one sample from this botnet can run at once on this machine, but, importantly, allows samples from other Zeus kit owner's botnets to coexist on the same machine.

Fig 6. Message box



Now the routine that will create and write the dropped file is decrypted. The encryption is byte-wise XOR with a rotating 4 byte key. The key and the size of the data to decrypt is obtained from the 0x200 byte block that was decrypted earlier. As we will see, this 200 byte block is replaced on the dropped file so the dropped file will not hold the key to decrypt this routine.

Execute the Dropping Routine

A randomly named registry key is created under "HKCU\Software\Microsoft". This is where the configuration data will be stored. A new 0x200 byte block is then constructed that will overwrite the existing block in the file.

This block will contain important information that the dropped file will need in order to successfully execute and includes the following:

- A string that identifies the infected machine – comprised of the computer name, OS version, OS install date, the OS DigitalProductId.
- A GUID identifying the drive that the file will be dropped to
- An RC4 encryption key – used to encrypt/decrypt the config data when it's written to and read from the registry, and to encrypt the stolen data.
- The path after %AppData% that the dropped file will reside at
- The name of the registry key under HKCU\Software\Microsoft

This block is written over the top of the old block and the file is written to the randomly generated path under %AppData% and given a filetime at a random date in the past.

Pass the Baton

The newly created file is then launched and a batch file is dropped and executed that will delete the dropper.

The Droppee

If we are running as the droppee then the first 0x1e6 bytes of the embedded 0x200 byte block is decrypted again, this time using a different RC4 key than used to check if the sample is the dropper or the droppee (but the same key that was used to encrypt the block when the dropper was creating the droppee).

Dropper	Droppee
Decrypt 0x200 byte block	Decrypt 0x200 byte block
Check DWORD value at offset 0x1E6	Check DWORD value at offset 0x1E6
Decrypt file creation routine	Decrypt first 0x1E6 bytes of block using different key
Write and execute new file	Verify block running on same system as created on
Write and execute self-deletion batch script	Inject into other processes
End	Continue ...

The same algorithm used to generate the GUID for the drive is executed and checked against the decrypted value, as is the pathname of the currently running executable against the values inside the decrypted block. If either of these checks fail then the program will terminate. This ensures the sample can only be run from the same location that it was dropped to.

The sample then writes its image into the address space and every process that it has permission to and a new thread is started inside that process. The main thread will then call ExitProcess and the focus of execution moves onto the injected threads.

The injected thread will first hook a large number of API's in the injected process. These API hooks are how Zbot intercepts and alters information flowing through the machine. The process name is then checked and if it is one of "dwm.exe", "taskhost.exe", "taskeng.exe", "wscntfy.exe", "ctfmon.exe", "rdpclip.exe", "explorer.exe" then a flag is set that indicates more threads will be launched from within this process (more on these later).

The injected thread will then go about stealing certain information that is stored on the victim's hard drive. This data includes:

- Data stored in browser cookies
- Any certificates that can be found using CertEnumCertificatesInStore().
- Data stored in flash cookies
- Credential information stored by the following FTP programs:
  - FlashFXPFTP
  - Total Commander
  - WSFTP
  - FilezillaFTP
  - FarManager
  - WinSCP
  - FTPCommander
  - CoreFTP
  - SmartFTP

In the most recent version (2.1) data is harvested from several other locations including from email programs such as Windows Mail and Outlook Express and from online poker application "Full Tilt Poker".

If the thread has been injected into one of the processes from the above list then several other threads are launched. These threads include:

- A thread that listens on various ports
- A thread that downloads and processes the configuration file
- A thread that monitors the bot's runkey entry in the registry, restoring it if it is removed

Configuration File Processing

The configuration file URL, along with the RC4 key to decrypt it, is encrypted and embedded in the Zbot binary. Once the thread responsible for downloading the config file has been started it will decrypt the region containing the URL and key and download the file over HTTP. It is then decrypted using the RC4 key and the MD5 of the decrypted file is computed. This is checked against a value inside the header of the decrypted configuration file to ensure the file is as expected and has not been corrupted in transit.

Fig 7. Snippet of code

```
loc_407434:                             ; CODE XREF: ReadConfigCh
push    10000000h
push    4E22h                           ; new exe file identifier
push    edi
xor     ebx, ebx
call    FindAndDecompressConfigItem
mov     ebx, eax
test    ebx, ebx
jz      short heapFree_done
lea     eax, [ebp+PathToExe]
push    eax                             ; lpFileName
push    (offset a_exe+2)                ; int
call    CreateNewTempFile
test    al, al
jz      short heapFree_heapFree_done
lea     esi, [ebp+var_40]
call    GetUserAgentString
mov     eax, hEvent
mov     [ebp+var_38], eax
lea     eax, [ebp+PathToExe]
push    0
mov     edi, esi
mov     [ebp+var_30], ebx
mov     [ebp+var_14], eax
call    CrackURLSendRequestInternetRead
test    al, al
jz      short deleteFile_heapFree_done
xor     eax, eax
push    eax                             ; int
push    eax                             ; int
push    eax                             ; int
movzx   eax, [ebp+arg_0]
neg     eax
sbb     eax, eax
and     eax, offset asc_402F9C          ; "~f"
push    eax                             ; AppDataDirPath
lea     eax, [ebp+PathToExe]
push    eax                             ; PathToExe
call    FormatExeStringCreateProcess
test    eax, eax
jz      short deleteFile_heapFree_done
mov     [ebp+RetVal??], 1
```

The thread will then re-encrypt the configuration data and write it into the registry (under the randomly named reg key under HKCU\Software\Microsoft that was created earlier). When written into the registry the configuration data is encrypted with a different key than was used to decrypt it after the download. The key used is inside the block that was written into the file by the dropper. Here is an overview of the process:

1. XOR decrypt region inside binary
2. Get config file URL and RC4 key1 from decrypted data
3. Download config file and use RC4 key1 to decrypt
4. Verify hash of decrypted data
5. Use RC4 key1 to decrypt block of data written into file by dropper
6. Get RC4 key2 from decrypted data
7. Re-encrypt data using RC4 key2 and write to registry

This second RC4 key is also used to encrypt stolen data when it is both stored temporarily on the disk and when it is sent back to the C & C server.

Once the configuration file has been downloaded and written into the registry, the same thread will attempt to download any new executable file that the configuration data points to.

The following is a snippet of the code that finds the new exe item and downloads the contents (Fig 7).

If there is an entry in the configuration file for an updated configuration file URL, the thread will also download and process that.

Behaviour Once Resident

Now we will move on to Zbot's general behaviour after it has infected a system.

API Hooks

Most of Zbot's data stealing logic is driven by the hooks it places inside processes. Here is a quick run down of typical Zbot API hooks:

The ntdll.dll hooks are intended to ensure that the Zbot memory resident component is injected into new processes and the new process's API's are hooked. Most of the rest are intended to monitor and steal data that those API's are used to send.

Dll Name	Api Name
ntdll.dll	NtCreateThread (pre Vista)
ntdll.dll	NtCreateUserProcess (Vista and later)
ntdll.dll	LdrLoadDll
kernel32.dll	GetFileAttributesExW
wininet.dll	HttpSendRequest
wininet.dll	HttpSendRequestEx
wininet.dll	InternetCloseHandle
wininet.dll	InternetReadFile
wininet.dll	InternetReadFileEx
wininet.dll	InternetQueryDataAvailable
wininet.dll	HttpQueryInfo
ws2_32.dll	closesocket
ws2_32.dll	send
ws2_32.dll	WSASend
user32.dll	OpenInputDesktop
user32.dll	SwitchDesktop
user32.dll	DefWindowProc
user32.dll	DefDlgProc
user32.dll	DefFrameProc
user32.dll	DefMDIChildProc
user32.dll	CallWindowProc
user32.dll	RegisterClass
user32.dll	RegisterClassEx
user32.dll	BeginPaint
user32.dll	EndPaint
user32.dll	GetDCEx
user32.dll	GetDC
user32.dll	GetWindowDC
user32.dll	ReleaseDC
user32.dll	GetUpdateRect
user32.dll	GetUpdateRgn
user32.dll	GetMessagePos
user32.dll	GetCursorPos
user32.dll	SetCursorPos
user32.dll	SetCapture
user32.dll	ReleaseCapture
user32.dll	GetCapture
user32.dll	GetMessage
user32.dll	PeekMessage
user32.dll	TranslateMessage
user32.dll	GetClipboardData
crypt32.dll	PFXImportCertStore
nspr4.dll	PR_OpenTCPSocket
nspr4.dll	PR_Close
nspr4.dll	PR_Read
nspr4.dll	PR_Write

Bot Behaviour

Zbot is also able to receive and act on several commands that can be issued by the bot master:

Command	Behaviour
os_shutdown	
os_reboot	
bot_uninstall	Re-download the config file and download any new exe it points to
bot_update	Add backdoor connection
bot_bc_add	Remove backdoor connection
bot_bc_remove	
bot_httpinject_disable	
bot_httpinject_enable	
fs_path_get	Implementation not present
fs_search_add	Implementation not present
fs_search_remove	Implementation not present
user_destroy	
user_logoff	
user_execute	Download and execute a file from a given URL
user_cookies_get	Steal data from cookies
user_cookies_remove	Delete cookie files (so that the user has re-enter login details)
user_certs_get	Steal digital certificates
user_certs_remove	Delete digital certificates
user_url_block	
user_url_unblock	
user_homepage_set	
user_ftpclients_get	Steal FTP credentials stored by FTP clients
user_flashplayer_get	
user_flashplayer_remove	

Emerging and Future trends

There have been several recent developments in the Zeus world that may dictate the future direction that Zbot takes.

Murofet, Domain Generation and File Infection

In the latter part of 2010 a major new revision of Zeus (2.1 in the binary and configuration file) was released.

Amongst the improvements two major features stood out:

The phone-home mechanism was updated to use a pseudo-random domain generator.

An extra hook was added to NtCreateFile in ntdll.dll that included functionality to infect files with '.exe' extensions as they were accessed.

The Domain Generation Algorithm is time based. The current time is used as a seed to cryptographically generate the full domain. This makes the Zeus botnet much more robust to take-down, as new C & C servers can be used as existing ones are taken offline, without having to update the executable on the infected machine.

The new code written to infected files uses the same domain generation algorithm that the Zbot executable uses to contact its command and control infrastructure. The infected files are generally known as Murofet. Each infected file has effectively become a downloader that will re-infect the system with the latest Zbot executable. This strategy increases the scope for infection and makes it more likely that a re-infection will occur if the original Zbot file is detected and removed.

SpyEye Merger

In late 2010 Brian Krebs [http://krebsonsecurity.com/2010/10/spyeye-v-zeus-rivalry-ends-in-quiet-merger/] reported that there was evidence to suggest on underground hacking forums that the Zeus kit author was retiring and had sold the source code to the author of rival crimeware kit SpyEye. A condition of sale was that the new owner continued to support existing Zeus customers.

There has certainly been no let-up on Zbot samples seen in the wild since this announcement so whatever the exact course of future development for Zeus, we are unlikely to have seen the back of it.

Conclusion

Zeus has grown into one of the most popular and widespread crimeware kits on the market. Its ease of use and effectiveness make it an attractive choice for today's cyber criminals.

A Zbot infection can be extremely costly. For an individual, theft of login details and online banking details can be disastrous. For an organization, the impact can be vastly more devastating.

The success of Zeus shows that this type of malware, be it in the form of Zeus itself or competitors to Zeus, is only likely to become even more widespread. Clearly, the demand for easy to use, information stealing Trojans is high, and as long as that demand exists there will be those who are willing to fulfil it.

Appendix

Configuration file encryption

Version 1:

The earliest configuration files used a simple fixed key encryption. These could be decrypted easily by anyone who knew the algorithm:

dataSize = size of data

dataIn = encrypted data

```
char b;
        for (i = 0; i < dataSize; i++)
        {
                dataOut[i] = 0;
        }
        for (i = 0; i < dataSize; i++)
        {
                b = dataIn[i];
                if ((i % 2) == 0)
                {
                b += 2 * i + 10;
                }
                else
                {
                b += 0xF9 - 2 * i;
                }
                dataOut[i] += b;
}
```

[threatexpert http://blog.threatexpert.com/2009/09/time-to-revisit-zeus-almighty.html]

Version 1.x:

The next major version saw each configuration file encrypted with RC4 using a 0x100 byte key generated at build time by the Zeus Builder. This key is unique to each botnet and is embedded in each bot executable at the same offset in the final section of the unpacked file.

Where *S is the 0x100 key byte array:

int rc4_decrypt(unsigned char *in, unsigned long size, unsigned char *S, unsigned char *out)

```
{
        int i, j, dataCount;
        i = j = dataCount = 0;
        unsigned char temp, rc4_byte;
        for (dataCount = 0; dataCount <
        size; dataCount++)
        {
                i = (i + 1) & 255;
                j = (j + S[i]) & 255;
                temp = S[j];
                S[j] = S[i];
                S[i] = temp;
                rc4_byte = S[(temp + S[j])
                & 255];

                out[dataCount] =
                in[dataCount] ^ rc4_byte;
        }
        return dataCount;
}
```

Version 2.0

This version saw an increase in obfuscation of the RC4 key and an extra level of encryption on top of RC4. This version also saw the configuration data stored in the registry rather than a file on the disk.

The easiest way to retrieve the RC4 key and URL to download the configuration file is from the Zbot PE file that is injected into running processes. Once the configuration file has been RC4 decrypted there is an extra XOR decryption on top as follows:

```
for (m = (decSize-1); m >0; m--)
        {
        decData[m] = decData[m]
        ^ decData[m-1];
        }
```

The decrypted configuration file consists of a header section then a number of blocks, each with their header indicating what should be done with them.

The header contains the following information:

- Size of the decrypted file
- Number of blocks
- Hash of the file

Each block has a header containing the following:

- An identifier field
- Another field used to indicate if the block is compressed or not
- Size of the data in the block compressed
- Size of the data uncompressed

If the data is compressed it is done so using unrv2b [http://qa.coreboot.org/docs/doxygen/src_2lib_2nrv2b_8c_source.html].

Stolen Data Encryption

Stolen data is stored temporarily in a file on disk before being transmitted back to the C & C server.

This temporary file starts with a DWORD value which is the length of the data chunk, XOR'ed with a key embedded in the Zbot PE file, followed by a Zero byte then the RC4 encrypted chunk of data. Then there is another XOR'ed DWORD, Zero byte, RC4'ed block of data and so on until the end of the file.

United Kingdom Sales:
Tel: +44 (0)8447 671131
Email: sales@sophos.com

North American Sales:
Toll Free: 1-866-866-2802
Email: nasales@sophos.com

Boston, USA | Oxford, UK

SOPHOS