

Alien Autopsy: Reverse Engineering Win32 Trojans on Linux

Joe Stewart 2002-11-14

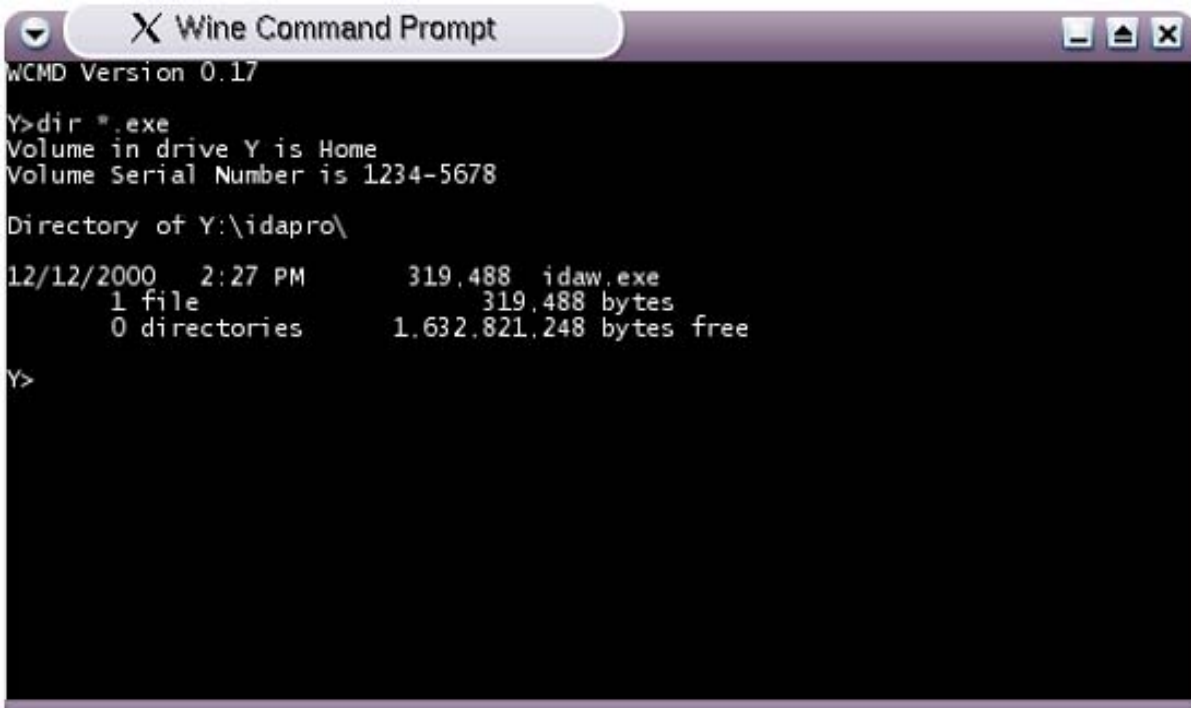
In my last article, [Reverse Engineering Hostile Code](#), I described the tools and processes involved in basic reverse engineering of a simple trojan. This article will offer a more detailed examination of the reversing process, using a trojan found in the wild. At the same time, this article will discuss some techniques for reversing Windows-native code entirely under Linux. As an added bonus, all the tools used in this article are either [freeware](#) or [free software](#). They are:

- [Wine](#) - the Win32 API implementation for Unix;
- [gdb](#) - our favorite Unix debugger and disassembly environment; and,
- [IDA Pro Freeware Version](#) - Win32 disassembler (runs on Linux under Wine release 20021007, may run under other versions as well).

Note: Readers who haven't read the previous article, [Reverse Engineering Hostile Code](#), may want to stop and do that now, unless they already have some knowledge of C and assembly language.

Getting a Deadlisting

A deadlisting is simply a dump of the assembly language code of the trojan. We will be using IDA Pro Freeware Version for this purpose. This program does a terrific job of cross-referencing jumps, calls and string data, and really makes the assembly source easy to read. The best part is it now works on Linux under Wine. Just [download the distribution](#) from DataRescue's site, and unzip it into an empty directory. Then change to the directory where you unzipped IDA and run the Wine utility *wcmd* from the programs subdirectory of the Wine source code.



```
Wine Command Prompt
WCMD Version 0.17
Y>dir *.exe
Volume in drive Y is Home
Volume Serial Number is 1234-5678

Directory of Y:\idapro\

12/12/2000  2:27 PM          319,488  idaw.exe
             1 file              319,488 bytes
             0 directories  1,632,821,248 bytes free

Y>
```

Figure 1 - wcmd, the command.com replacement for Wine

A new console window should open as shown in figure 1. This is an environment similar to CMD.EXE or COMMAND.COM on Windows. Since you started wcmd while in the idapro directory, you now need to simply type "idaw.exe" into the wcmd window and press enter. If successful, you should see the wcmd screen change as shown in figure 2. You are now running IDA Pro under Linux.

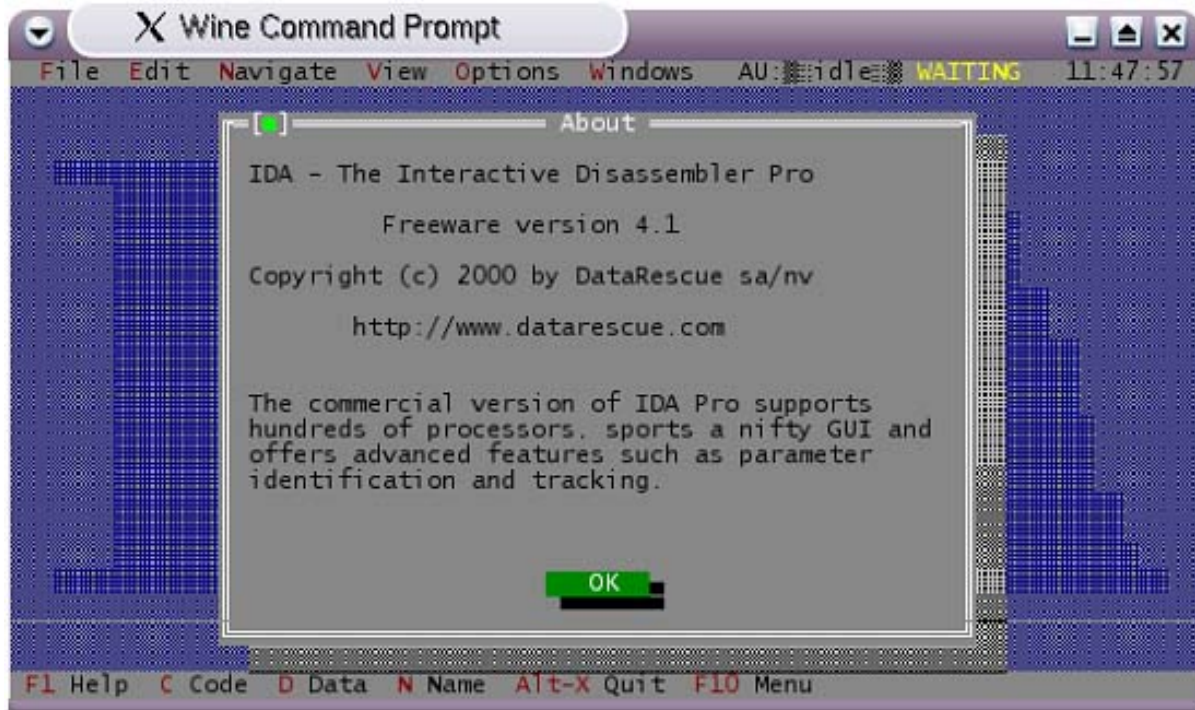


Figure 2 - IDA Pro Freeware Version, running under Wine

If you click OK, you will see the file chooser dialog. Browse the directory tree looking for the Win32 executable you want to disassemble and click OK. Next, it will ask for some file format options. The defaults are usually sane, so you can click OK on this dialog. IDA will go to work processing all the information it finds in the executable format, and will show a "thinking" indicator in the toolbar while it is working. It can take a few seconds to several minutes to run, depending on the size of the file.

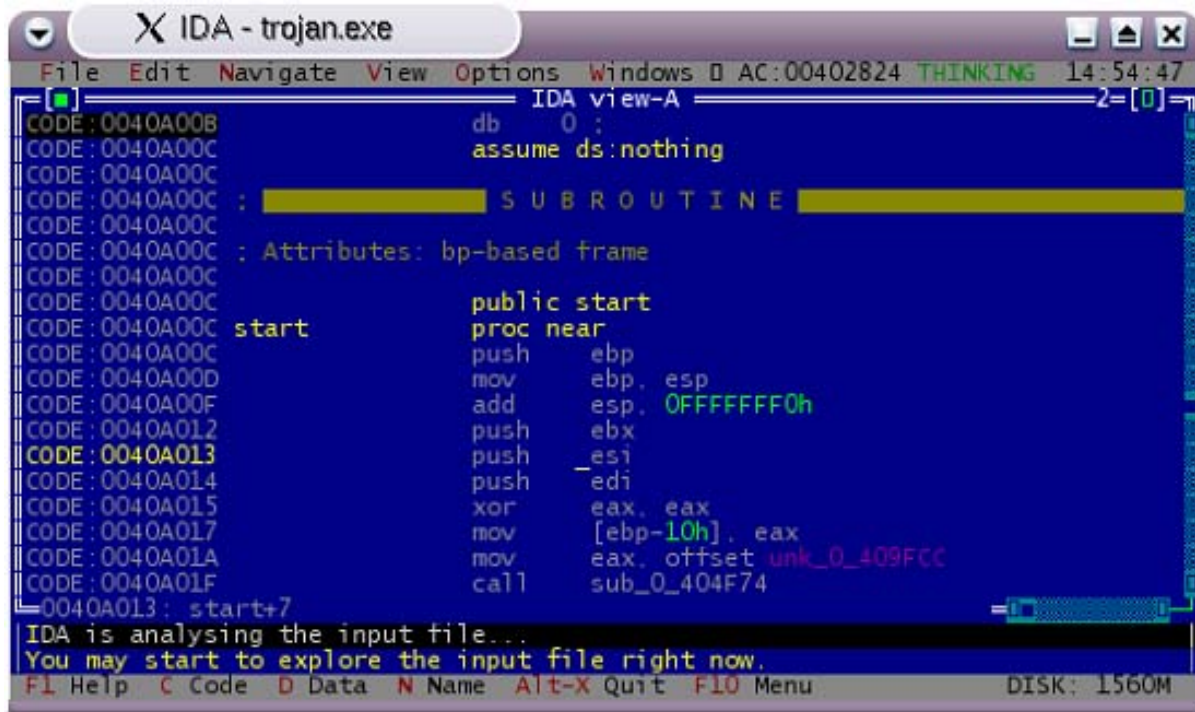


Figure 3 - IDA analyzing the code

After IDA finishes its analysis, select "File|Produce output file}Produce LST file" from the menus and save the file to disk. You now have a deadlisting of the Win32 PE file.

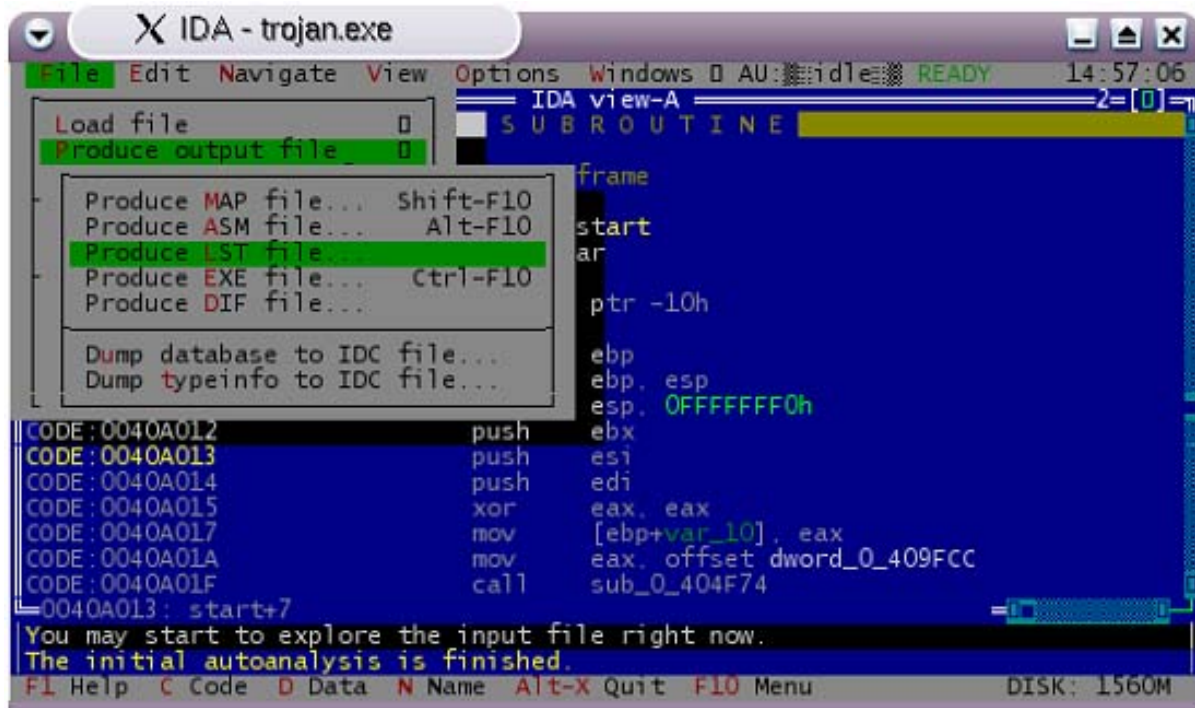


Figure 4 - Saving the deadlisting

Reading the Deadlisting

Open the .LST file you saved in any text editor, and find the start of the program. It is usually not at the top of the

file; you can search for "start" or sometimes "WinMain". In this trojan, execution begins at offset 0x40A00C. This is where we will begin reading through the code. The main clues available in the deadlisting are Win32 API calls (which IDA Pro nicely labels for us) and offsets to string data. Here is the start of the main subroutine, all the way to the first API call:

```

CODE:0040A00C
CODE:0040A00C ; 0000000000000000 S U B R O U T I N E 000000000000000000000000
CODE:0040A00C
CODE:0040A00C ; Attributes: bp-based frame
CODE:0040A00C
CODE:0040A00C          public start
CODE:0040A00C start          proc near
CODE:0040A00C
CODE:0040A00C var_10          = dword ptr -10h
CODE:0040A00C
CODE:0040A00C          push    ebp
CODE:0040A00D          mov     ebp, esp
CODE:0040A00F          add     esp, 0FFFFFFF0h
CODE:0040A012          push   ebx
CODE:0040A013          push   esi
CODE:0040A014          push   edi
CODE:0040A015          xor    eax, eax
CODE:0040A017          mov    [ebp+var_10], eax
CODE:0040A01A          mov    eax, offset dword_0_409FCC
CODE:0040A01F          call  sub_0_404F74
CODE:0040A024          mov    edi, offset unk_0_40C7B8
CODE:0040A029          xor    eax, eax
CODE:0040A02B          push   ebp
CODE:0040A02C          push   offset loc_0_40A15E
CODE:0040A031          push   dword ptr fs:[eax]
CODE:0040A034          mov    fs:[eax], esp
CODE:0040A037          push   offset unk_0_40C604
CODE:0040A03C          push   2
CODE:0040A03E          call  j_WSASStartup

```

j_WSASStartup is a local subroutine which acts as a wrapper for the [Win32 API call](#). Given this, we know this program is likely to be opening a socket for network communication.

```

CODE:0040A043          push   6
CODE:0040A045          push   1
CODE:0040A047          push   2
CODE:0040A049          call  j_socket
CODE:0040A04E          mov    ebx, eax

```

```

CODE:0040A050      mov     ds:word_0_40C794, 1
CODE:0040A059      mov     ds:word_0_40C796, 0
CODE:0040A062      push   4
CODE:0040A064      push   offset word_0_40C794
CODE:0040A069      push   80h
CODE:0040A06E      push   0FFFFh
CODE:0040A073      push   ebx
CODE:0040A074      call   j_setsockopt

```

Two more wrapped API calls, [setsockopt](#). We're going to be looking up Win32 API calls every step of the way, so let's discuss resources in this area. There are some [good books](#) on the subject, but you may find that the online, searchable [MSDN Documentation](#) is a better resource to have while deep inside a deadlisting.

Looking up the *socket* function prototype, we see that programs must prepare the call as follows:

```
SOCKET socket(int af, int type, int protocol);
```

At the assembly level, these arguments are passed to the system by pushing them onto the stack, last argument first. Look at the lines we found starting at offset 0x40A043 above, just prior to the *j_socket* call:

```

CODE:0040A043      push   6
CODE:0040A045      push   1
CODE:0040A047      push   2
CODE:0040A049      call   j_socket

```

So, when calling *socket*, the variable *af* would have a value of 2, *type* would have a value of 1, and *protocol* would have a value of 6. Referring to the [MSDN documentation](#) we see that an *af* value of 2 means AF_INET, or Internet address family. A type 1 socket is defined as SOCK_STREAM, and protocol 6 is TCP. So our trojan is establishing a TCP socket. We still don't know on what port the socket is to be opened, and if it is to be a listener or a client. Continue reading through the deadlisting.

```

CODE:0040A074      call   j_setsockopt
CODE:0040A079      mov     ds:word_0_40C798, 2
CODE:0040A082      call   sub_0_402700
CODE:0040A087      dec     eax
CODE:0040A088      jnz    short loc_0_40A0A5
CODE:0040A08A      lea    edx, [ebp+var_10]
CODE:0040A08D      mov     eax, 1
CODE:0040A092      call   sub_0_402760
CODE:0040A097      mov     eax, [ebp+var_10]
CODE:0040A09A      call   sub_0_405EA0
CODE:0040A09F      mov     ds:word_0_40B2C0, ax
CODE:0040A0A5

```

```

CODE:0040A0A5  loc_0_40A0A5:                ; CODE XREF: start+7C^Xj
CODE:0040A0A5          mov     ax, ds:word_0_40B2C0
CODE:0040A0AB          push   eax
CODE:0040A0AC          call   j_htons
CODE:0040A0B1          mov     ds:word_0_40C79A, ax
CODE:0040A0B7          push   10h
CODE:0040A0B9          push   offset word_0_40C798
CODE:0040A0BE          push   ebx
CODE:0040A0BF          call   j_bind

```

The call to [bind](#) is prototyped as follows:

```
int bind(SOCKET s, const struct SOCK_ADDR* name, int namelen);
```

So at this point, the EBX register contains a pointer to the socket, and the data segment offset 0x40C798 should contain our SOCK_ADDR structure which is 16 bytes long (10h). The SOCK_ADDR structure for TCP is defined as:

```

struct sockaddr_in {
    short sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};

```

The port number is contained in this structure (sin_port). It is in network byte-order (big endian), so if the port was stored in a variable, it had to be converted from the x86's normal host byte-order (little-endian). Sure enough, if we look back to offset 0x40A0AC, we see the API call [htons](#), which converts a 16-bit short integer from host order to network order. So the argument to this call (EAX at 0x40A0AC) is our port number. Let's jump back and look at the code just prior to the *htons* call, in order to see how our port variable got filled:

```

CODE:0040A09A          call   sub_0_405EA0
CODE:0040A09F          mov     ds:word_0_40B2C0, ax
CODE:0040A0A5          mov     ax, ds:word_0_40B2C0
CODE:0040A0AB          push   eax
CODE:0040A0AC          call   j_htons

```

Tracing the origin of the value in EAX back from 0x40A0AC shows us it was likely a return value from subroutine 0x405EA0 called at 0x40A09A. This means we'll have to trace back at least one subroutine, maybe more, to find the port number. But we don't care to do all that work, so we will get that information another way. For now, let's return to where we were and continue on:

```
CODE:0040A0C4
```

```

CODE:0040A0C4 loc_0_40A0C4:                ; CODE XREF: start+125^Yj
CODE:0040A0C4                push    5
CODE:0040A0C6                push    ebx
CODE:0040A0C7                call   j_listen
CODE:0040A0CC                mov     dword ptr [edi], 10h
CODE:0040A0D2                push    edi
CODE:0040A0D3                push    offset unk_0_40C7A8
CODE:0040A0D8                push    ebx
CODE:0040A0D9                call   j_accept

```

At this point, seeing the `accept` call tells us we are going to be a listener. Now we would like to know on what port.

Running the Code: Wine+gdb

The Wine project has ported a large set of the Win32 API calls to Linux. This enables us to run our trojan under Linux, and use gdb to debug it as if it were a native Linux executable. Wine has its own debugger, which could also serve the same purpose; however, gdb is more feature-rich. The only drawback to using gdb is that you must first trace your way through the Wine code, instead of working immediately on the Win32 code. I have a couple of breakpoints I will share below that can quickly get you past the Wine routines to land at the start of the Win32 process.

Don't forget the basic rules of reversing unknown/hostile code. Always work on non-production machines, on a network segment specifically designed to contain the trojan's traffic. Now that the safety lecture is over, launch gdb with the path to the Wine executable (not the trojan) as an argument. On my system, I would run the following command:

```
gdb /usr/bin/wine.bin
```

This would produce the following output. (Commands to enter after starting gdb are shown below in bold; comments in red are the author's notes)

```

GNU gdb 5.2.1-2mdk (Mandrake Linux)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i586-mandrake-linux-gnu"...(no debugging symbols found)...
(gdb) set args ./trojan.exe

```

(Give gdb the argument to feed to Wine; the name of the trojan to load)

```
(gdb) b PROCESS_InitWine
```

(Set a breakpoint when Wine begins. This is to allow us to set breakpoints later on which are currently inaccessible since the code has not been loaded into memory yet)

Breakpoint 1 at 0x80486d0

(gdb) run

(Run the program)

Starting program: /usr/bin/wine.bin ./trojan.exe

(no debugging symbols found)...(no debugging symbols found)...

(no debugging symbols found)...(no debugging symbols found)...

(no debugging symbols found)...(no debugging symbols found)...

Breakpoint 1, 0x400eea46 in PROCESS_InitWine () from /usr/lib/libntdll.dll.so

(gdb) b SYSDEPS_SwitchToThreadStack

(Set a breakpoint at beginning of Win32 program execution. After this breakpoint is reached we can set our target breakpoint at 0x40a0ab)

Breakpoint 2 at 0x400f4d66

(gdb) c

(Continue executing the program)

Continuing.

Could not stat /floppy (No such file or directory), ignoring drive A:

Breakpoint 2, 0x400f4d66 in SYSDEPS_SwitchToThreadStack () from /usr/lib/libntdll.dll.so

(gdb) b * 0x0040a0ab

(Set a breakpoint at the offset just prior to the htons API call. When setting a breakpoint on an offset, don't forget to prefix the offset with an asterisk)

Breakpoint 3 at 0x40a0ab

(gdb) c

(Continue executing the program)

Continuing.

Breakpoint 3, 0x0040a0ab in ?? ()

(We've hit our final breakpoint and landed in our trojan's main subroutine, just prior to the htons call. EAX should now contain our port number)

```
(gdb) print/x (short) $eax
```

(Print out the value in the EAX register)

```
$1 = 0x4b8c
```

(Our port number, in little-endian hexadecimal. 0x4b8c converted to decimal = 19340)

```
(gdb) quit
```

Without tracing through countless subroutines in the deadlisting we have determined the trojan would have opened a listener on port 19340.

Coroner's Report

At this point, we could continue reading the deadlisting and filling in the blanks in our knowledge by setting breakpoints and reading variables/memory in gdb until every function of the program is exposed. Obviously this short article can't cover every function of even this very small trojan, but hopefully it has given you enough information to strike out on your own and conquer new trojans with a little bit of assembly, a little bit of C, a little bit of luck and a lot of patience. At first you may feel overwhelmed when presented with thousands upon thousands of lines of assembly code, but it does get easier with experience. You will eventually find patterns in the larger picture and become more removed from the line-by-line analysis, almost as if you were "feeling" the code rather than just reading it.

About the Trojan

The code above is from a Win32 backdoor found on a compromised system. Connecting to the listener produces a banner of "Barvinok NT Shell v1.0" and a password prompt. The password is hard-coded in the binary and readable in a "strings" output. After entering the password you have a basic NT command-line shell.

Joe Stewart is a Senior Information Security Analyst with [LURHQ Corporation](#), a Managed Security Services Provider located in Myrtle Beach, South Carolina.

[Privacy Statement](#)

Copyright 2006, SecurityFocus