# Fighting EPO Viruses

*Piotr Bania*,

This short article describes the so-called Entry-Point Obscuring (EPO) virus coding tech
primarily through a direct analysis of the Win32.CTX.Phage virus. The reader should k
basics of IA-32 assembly and the main elements of the Portable Executable (PE) file st
fully understand this article. The author also advises the reader to review the Win32.C
description written by Peter Szor and Wason Han , since this article does not cover all
of the virus.

### Why EPO and Win32.CTX.Phage

Entry-point obscuring viruses are very interesting because of the very difficult nature of its det
disinfection and removal. Nowadays the EPO technique is used in many different ways, howeve
Win32.CTX.Phage has been chosen for this article because it was written by the same author o
infamous viruses as Win9x.Margburg (one of the first Windows9x polymorphic virus, which first
the wildlist) and Win9x.HPS. The author of these viruses is known for his difficult-to-detect and
disinfect creations. CTX.Phage in particular involves many techniques that make the disinfectio
highly difficult, even after the virus is fully understood.

### Understanding the Entry-Point Obscuring (EPO) technique

When a virus infects a file, it must find some way to attain control and be executed. Most of th
infectors use the most common way of doing this -- they simply change the entry-point of the
application and make it point to the virus body. An example is shown below.

| Original EXE | Infected EXE |
|---|---|
| Entry-point: **0x1000 (.code section)** | Entry-point: **0x6000 (.reloc section)** |

Such virus activity is very easy to detect, as it usually results in files whose entry-point resides
code section, and are therefore marked as suspicious by a virus scanner. Here is some exampl
detects this type of infection:

```
(checks if the 'entry-point section' is the last section):

// --- snip of scanner code ------------------------------------------------
...(snip)...
sections = pPE->FileHeader.NumberOfSections;
pSH = (PIMAGE_SECTION_HEADER)((DWORD)mymap+pMZ->e_lfanew + sizeof(IMAGE_NT_HEADERS));


    while (sections != 0) {
        if (IsBadReadPtr(&pSH,sizeof(PIMAGE_SECTION_HEADER)) == TRUE)
              {
            printf("[-] Error: Bad PE file\n");
            goto error_mode4;
        }

        char *secname=(char *) pSH->Name;
        if (secname == NULL) strcpy(secname,"NONAME");

        startrange=(DWORD) pSH->VirtualAddress + pPE->OptionalHeader.ImageBase;
        endrange=(DWORD) startrange + pSH->Misc.VirtualSize;

        ...(snip)...

        if (pSH->VirtualAddress <= pPE->OptionalHeader.AddressOfEntryPoint && \
                    pPE->OptionalHeader.AddressOfEntryPoint < pSH->VirtualAddress +
```

```
                                                pSH->Misc.VirtualSize)
                  {

                      printf("[+] Checking call/jump requests from %s section (EP)\n",
                                                        secname);
            pSHC = pSH;
        }


        pSH++;
        sections--;
    }

    pSH--;

    if (pSHC == NULL)
       {
        printf("[-] Error: invalid entrypoint\n");
        goto error_mode4;
    }


    printf("[+] Starting heuristics scan on %s section...\n\n",pSHC->Name);

    if (pSHC == pSH)
       {
        printf("[!] Alert: Entrypoint points to last section (%s) -> 0x%.08x\n",
                           pSH->Name,pPE->OptionalHeader.AddressOfEntryPoint +
                               pPE->OptionalHeader.ImageBase);

        printf("[!] Alert: The file may be infected!\n");
        printf("[+] No deep-scan action was performed\n");
        goto error_mode4;
    }


...(snip)...
// --- snip of scanner code --------------------------------------------
```

The very reason why the EPO technique was developed was to avoid virus scanner detection. A
obscuring virus is a virus that doesn't get control from the host program directly. Typically, the
the host program with a jump/call routine, and receives control that way. While there are many
the EPO technique, in this article we will look at one of them in detail.

### The EPO technique used in Win32.CTX.Phage

The Phage virus doesn't modify the entry-point of an infected file, instead it scans all over the h
section and searches for API calls generated by Borland or the Microsoft linker. When such code
virus checks that the destination address points somewhere inside the IMPORT section. If the c
import call, Phage gets a random number which tells the virus to patch the current processed i
find next one. Figures 1, 2, 3, and 4 below show a few example schemas.

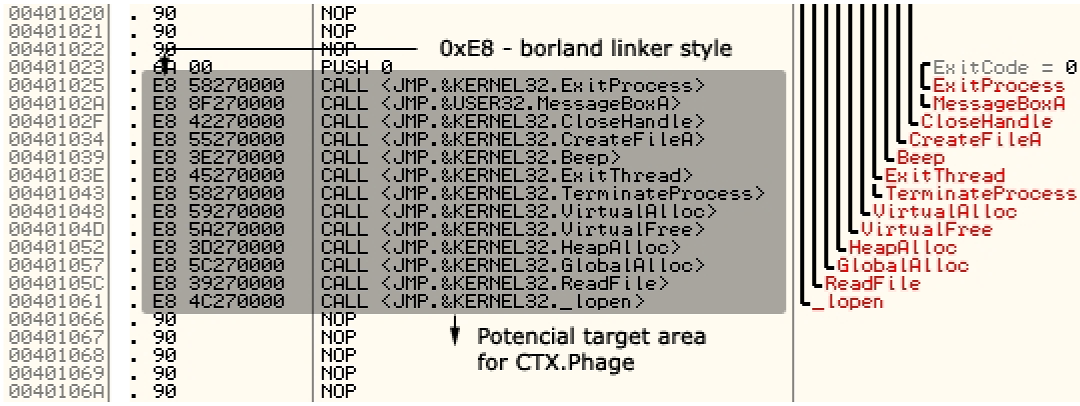

Figure 1. Original application (ENTRYPOINT: 0x1000 – LINKER: BORLAND).

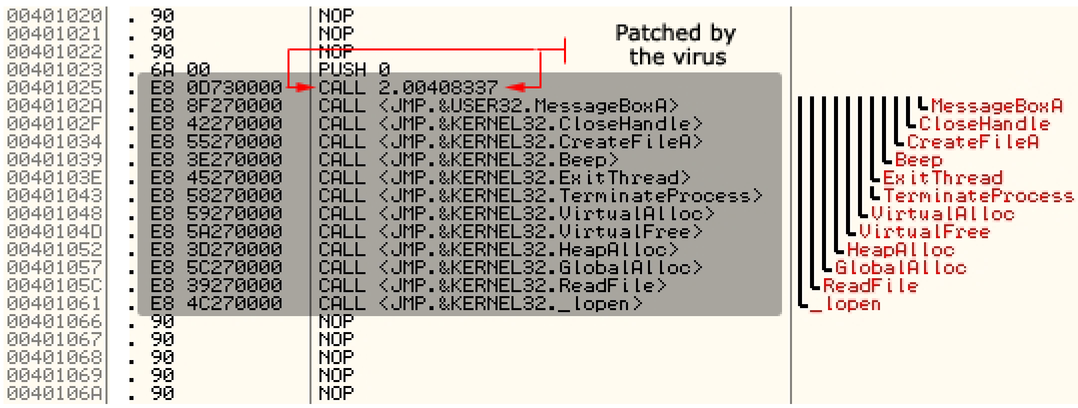

Figure 2. Infected application (ENTRYPOINT: 0x1000 – LINKER: BORLAND).

Figure 3. Original application (ENTRYPOINT: 0x1039 – LINKER: MICROSOFT).

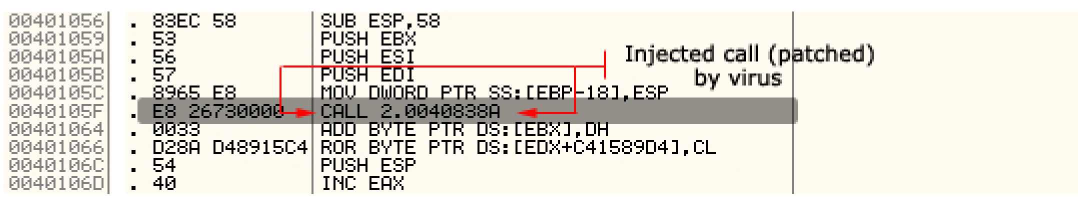

Figure 4. Infected application (ENTRYPOINT: 0x1039 – LINKER: MICROSOFT).

The above schemas show how the CTX.Phage EPO virus works. As mentioned before, the virus instruction by overwriting it with a randomly found call. As the application size grows (and also call range from the entry-point), it becomes increasingly difficult to find the injection of the viru other hand, while using this EPO technique reduces the risk of virus execution, there are also s when the "call-to-virus" will not be executed at all.

At this point, let's find a way to detect such injections such that it does not cause false alarms.

### Finding the virus injection

How difficult is it to find CTX.Phage injections? First of all, the virus inserts a call instruction as

| | |
|---|---|
| **E8 ?? ?? ?? ??** | **CALL XXXXXXXX** |

Where:

- **E8** is the CALL instruction opcode
- **?? ?? ?? ??** is the instruction operands (destination)

Before we go any further, let's summarize all the information we know about the current EPO:

1. The injection is always done somewhere behind the entry-point.
2. The injected call executes the virus code which is stored always in last section (this bit of really helpful).

As the reader probably knows, we could simply search for 0xE8 bytes (call opcodes) but there i possibility that we might find some "suspicious" call that thands in non-call instruction, for exar

| | |
|---|---|
| **68 332211E8** | **PUSH E8112233** |

As you can see, this is the push instruction, but the scanner finds the E8 byte and could consid Unless we don't want to build up our disassembler engine (which is very long and hard work) w another way. Yes, you guessed it: we need to add a condition for the E8 byte scanning routine, that the call always executes code that resides in last section! Now that everything is clear, her conditions we require:

```
temp_loc = (DWORD)((DWORD)pSHC->VirtualAddress + i + (*(DWORD*)loc)) + 5;
if (temp_loc >= pSH->VirtualAddress && temp_loc <= pSH->VirtualAddress + pSH-
>Misc.VirtualSize) BAD_CALL = 1;
```

Where:

- **temp_loc** is the calculated destination of found call (E8 opcode)
- **pSH** is the header of last section
- **+ 5** is the size of call instruction (opcode + destination)

A sample temp_loc calculation might look as follows:

```
Scanned instruction:
00401025  \. E8 58270000    CALL

Calculation:
temp_loc = 1025 (virtual address) + 00002758 (call destination) + 5 (size of call instru
```

If the temp_loc address resides somewhere between last section's virtual address (start) and tl section's virtual address + its virtual size, the call is marked as suspicious. Here is the short sn author's scanner:

```
(searches for call and jump instructions and checks theirs destinations):

// --- snip of scanner code ------------------------------------------------
...(snip)...

printf("[+] Starting from offset: 0x%.08x\n",pPE->OptionalHeader.ImageBase +
    pSHC->VirtualAddress);
```

```
    for (i = 0; (i != pSHC->SizeOfRawData); i++)
    {
        loc = (DWORD)((DWORD)mymap + pSHC->PointerToRawData) + i;

        if ((*(BYTE*)loc) == O_CALL || (*(BYTE*)loc) == O_JMP )
        {
            loc++;
            temp_loc = (DWORD)((DWORD)pSHC->VirtualAddress + i + (*(DWORD*)loc)) + 5;

            if (temp_loc >= pSH->VirtualAddress && temp_loc <= pSH->VirtualAddress + \
                          pSH->Misc.VirtualSize)
            {
                printf("[!] Alert: Detected request to %s(0x%.08x) section at: 0x%.08x\n
                                        pSH->Name,pPE->OptionalHeader.ImageBase + temp_l
                    pSHC->VirtualAddress + pPE->OptionalHeader.ImageBase + i);

                if (where_ctx == NULL)
                {
                    where_ctx = (DWORD)(pPE->OptionalHeader.ImageBase + temp_loc);
                    caller = (DWORD)(pSHC->VirtualAddress + \
                                                   pPE->OptionalHeader.ImageBase + i);

                    upa = (DWORD)(pSH->VirtualAddress + pPE->OptionalHeader.ImageBase);

                    sv = loc - 1;

                }
                count++;
            }
            loc--;
        }

    }

    printf("[+] Scan finished, %d suspected instruction(s) found.\n",count);

...(snip)...
// --- snip of scanner code ---------------------------------------------
```

While scanning files with this code, I haven't seen any false alarms, so it is probably one of the
or techniques one can use to find such virus injections.

### Clearing the code, deleting the injection

Since our scanner is able to find the injected call, we can move on. Now we need to reload the
On other words, we need to clear the injection. To do this we should first know more informatic
virus.

1. The injected call flows the execution to a polymorphic decryptor, which is generated in a
   several decryption phases can occur, from 4 to 7.
2. The virus must reset the "hooked" call before returning the execution to the host. Otherw
   infected application may fault. The original instruction is saved somewhere inside the viru

The main problem is that the virus is encrypted and the polymorphic decryptor will decrypt the
several times. We need to obtain the clear virus body in order to reset the original instruction.
those bytes directly since the code is encrypted. There are a couple of solutions to clear/bypass
polymorphic decryption layers, such as using emulation and so on. Writing a full emulator is su
quick and easy job, however a different solution does exist. Most Windows viruses use the GetP
API to obtain needed API addresses for their future execution. Lets try to set a breakpoint at G
(of course to avoid false GetProcAddress requests. First we need to execute the virus injection,
since we have located it before). This is shown below in Figure 5.

```
0012FF84   00406AF9 ┌CALL to GetProcAddress from 2.00406AF3
0012FF88   77E60000 │hModule = 77E60000 (kernel32)
0012FF8C   77ECFAD0 └ProcNameOrOrdinal = "CreateFileA"
```

Figure 5. GetProcAddress.

The call came from 0x406AF3, which in fact points to the decrypted body. Indeed, the poly lay bypassed! Here is the sample proof using the decrypted string, shown in Figure 6.

```
00406149 C3 5B 20 20 43 54 58 20 50 68 61 67 65 20 56 69 ├[  CTX Phage Vi
00406159 72 75 73 20 42 69 6F 43 6F 64 65 64 20 62 79 20 rus BioCoded by 
00406169 47 72 69 59 6F 20 2F 20 32 39 41 20 20 44 69 73 GriYo / 29A  Dis
00406179 63 6C 61 69 6D 65 72 3A 20 54 68 69 73 20 73 6F claimer: This so
00406189 66 74 77 61 72 65 20 68 61 73 20 62 65 65 6E 20 ftware has been 
00406199 64 65 73 69 67 6E 65 64 20 66 6F 72 20 72 65 73 designed for res
004061A9 65 61 72 63 68 20 70 75 72 70 6F 73 65 73 20 6F earch purposes o
004061B9 6E 6C 79 2E 20 54 68 65 20 61 75 74 68 6F 72 20 nly. The author 
004061C9 69 73 20 6E 6F 74 20 72 65 73 70 6F 6E 73 69 62 is not responsib
004061D9 6C 65 20 66 6F 72 20 61 6E 79 20 70 72 6F 62 6C le for any probl
004061E9 65 6D 73 20 63 61 75 73 65 64 20 64 75 65 20 74 ems caused due t
004061F9 6F 20 69 6D 70 72 6F 70 65 72 20 6F 72 20 69 6C o improper or il
00406209 6C 65 67 61 6C 20 75 73 61 67 65 20 6F 66 20 69 legal usage of i
00406219 74 20 20 5D E8 00 00 00 00 5D 81 ED 22 32 40 00 t  ]R....]üY"2@.
```

Figure 6. Decrypted string.

To make the disinfector able to break on GetProcAddress, we need to build a small debugger ( the fastest way to do it). This is easy since Windows platform already comes with Debug APIs.

Basically, following the code debugs the virus process, modifies the original entry of GetProcAd (nop), 0x90 (nop), 0xCC (int 3 – breakpoint) and takes over the EXCEPTION_BREAKPOINT onl from the "hooked" range:

```
(debugs process, executes virus call, hooks GetProcAddress and obtains caller (virus) ac

// --- snip of scanner code ------------------------------------------------
...(snip)...
unsigned char patch[4] = { 0x90, 0x90, 0xCC };
_GetProcAddress = (DWORD) GetProcAddress(LoadLibrary("KERNEL32.DLL"), "GetProcAddress");

GetStartupInfo(&si);
if (!CreateProcess(NULL,temp_name,NULL,NULL,FALSE,DEBUG_PROCESS +
    DEBUG_ONLY_THIS_PROCESS, NULL, NULL, &si, π))
{
    printf("[-] Error: cannot create process, error: %d\n",GetLastError());
    goto error_di;
}

printf("\n[+] Process created, pid=0x%.08x\n",pi.dwProcessId);
printf("[+] Starting emulation engine...\n");

while (1)
{
    WaitForDebugEvent(&de,INFINITE);
    if (de.dwDebugEventCode == EXIT_PROCESS_DEBUG_EVENT) {
        printf("[!] Error: ups process exited...\n");
        goto error_term;
    }

    if (de.dwDebugEventCode == EXCEPTION_DEBUG_EVENT)
    {
        if (de.u.Exception.ExceptionRecord.ExceptionCode == EXCEPTION_ACCESS_VIOLATION)
            if (de.u.Exception.dwFirstChance == TRUE)
            {
                printf("[+] Exception occured at: 0x%.08x, passing to
                                        program.\n",de.u.Exception.ExceptionRecord.Excep

                ContinueDebugEvent(de.dwProcessId,de.dwThreadId,\
                    DBG_EXCEPTION_NOT_HANDLED);
                }
                else
                {
                    printf("[-] Hard error occured, terminating the program\n");
```

```
            printf("[-] Disinfecting failed\n");
            goto error_term;
        }

    }

    if (de.u.Exception.ExceptionRecord.ExceptionCode == EXCEPTION_BREAKPOINT)
    {
        if (fe == NULL)
        {
            fe = 1;
            printf("[+] Reached break point at 0x%.08x\n",
                                    de.u.Exception.ExceptionRecord.Exceptio

            printf("[+] Modifing 4 bytes at host stack\n");


            tc.ContextFlags = CONTEXT_CONTROL;
            if (!GetThreadContext(pi.hThread, &tc))
            {
                printf("[-] Failed to get thread context, error: %d\n",
                                                 GetLastError());
                printf("[-] Disinfecting failed\n");
                goto error_term;
            }

            ReadProcessMemory(pi.hProcess, (void*)tc.Esp, &stack_v,4,NULL);

            if (stack_v == NULL)
            {
                printf("[-] Error: reading from stack failed\n");
                printf("[-] Disinfecting failed\n");
                goto error_term;
            }

            tc.Esp = tc.Esp - 4;
            caller += 5;

            if (!WriteProcessMemory(pi.hProcess, (void*)tc.Esp, &caller, 4,
                                     NULL))
            {
                printf("[-] Error: writing to stack failed\n");
                printf("[-] Disinfecting failed\n");
                goto error_term;
            }
            printf("[+] Stack modified, 0x%.08x added caller -> 0x%.08x\n", \
                tc.Esp, caller);


            printf("[+] Redirecting EIP to 0x%.08x...\n",where_ctx);
            tc.Eip = where_ctx;

            if (!SetThreadContext(pi.hThread, &tc))
            {

                printf("[-] Failed to set thread context, error: %d\n", \
                    GetLastError());

                printf("[-] Disinfecting failed\n");
                goto error_term;

            }

            VirtualProtectEx(pi.hProcess, (void*) _GetProcAddress, sizeof(patch)
                                     PAGE_READWRITE, &oldp);

            WriteProcessMemory(pi.hProcess, (void*) _GetProcAddress, &patch,
                                     sizeof(patch), NULL);

            VirtualProtectEx(pi.hProcess, (void*) _GetProcAddress, sizeof(patch)
```

```
                                                    oldp, &oldp);

                    printf("[+] Placed breaker at 0x%.08x\n",_GetProcAddress);


                    ContinueDebugEvent(de.dwProcessId,de.dwThreadId,DBG_CONTINUE);
                }


                if ((DWORD) de.u.Exception.ExceptionRecord.ExceptionAddress >
                    _GetProcAddress && (DWORD) de.u.Exception.ExceptionRecord.ExceptionA
                                                 < _GetProcAddress + sizeof(patch))
                {

                    printf("[+] Virus reached the breaker at 0x%.08x\n", \
                        de.u.Exception.ExceptionRecord.ExceptionAddress);


                    tc.ContextFlags = CONTEXT_CONTROL;
                    if (!GetThreadContext(pi.hThread, &tc))
                    {
                        printf("[-] Failed to get thread context, error: %d\n", \
                                                   GetLastError());

                        printf("[-] Disinfecting failed\n");
                        goto error_term;
                    }

                    ReadProcessMemory(pi.hProcess, (void*)tc.Esp, &stack_v, 4, NULL);
                    printf("[+] Virus request captured from 0x%.08x\n",stack_v);
                                 ...(snip)...


...(snip)...
        ContinueDebugEvent(de.dwProcessId,de.dwThreadId,DBG_EXCEPTION_NOT_HANDLED);

...(snip)...
// --- snip of scanner code ---------------------------------------------------
```

Now when we have the clean virus body we can try to locate the original instructions. Since CT doesn't modify the bits from the host code section, it has only one way to reset the original ins using WriteProcessMemory API (well, it could use VirtualProtect API to get write access to host and then write the original bytes, but it doesn't). So here is the break on WriteProcessMemory, Figure 7.

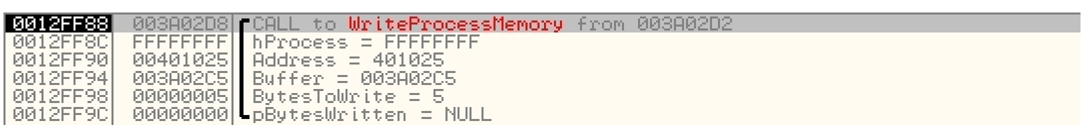

Figure 7. Break on WriteProcessMemory.

As you can see, BytesToWrite is equal to 5 and Address is equal to the location found by the sc only problem is that the call comes from allocated memory (the virus allocated it, copied itself execution from there). But lets try to check the caller address below in Figure 8.

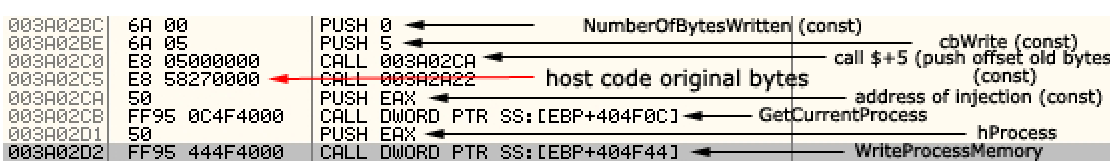

Figure 8. Checking the caller address.

The "const" bytes (for example those marked in the picture above) are:

**6A 00 6A 05 E8 05 00 00 00 ?? ?? ?? ?? ?? 50**

Where:

- **6A 00** is push 0
- **6A 05** is push 5
- **E8 05 00 00** is call $+5
- **?? ?? ?? ?? ??** is the original host bytes (wildcard)
- **50** is push eax

Here is the signature, useful to find original host bytes (there are the same in every generation these ones are located in the allocated memory. So the question is: does the same bytes exists inside the unencrypted body of virus, in other words, somewhere inside last section? Lets try to Figure 9.

```
004062BC | 6A 00          | PUSH 0
004062BE | 6A 05          | PUSH 5
004062C0 | E8 05000000    | CALL 2.004062CA
004062C5 | E8 58270000    | CALL 2.00408A22
004062CA | 50             | PUSH EAX
004062CB | FF95 0C4F4000  | CALL DWORD PTR SS:[EBP+404F0C]
004062D1 | 50             | PUSH EAX
004062D2 | FF95 444F4000  | CALL DWORD PTR SS:[EBP+404F44]
```

Figure 9. Scanning the virus.

Indeed, the same bytes were found in "native" virus location. The GetProcAddress was called b from 0x406AF3, as you can see the original bytes that lay far before it. Here is the code examp scanner which searches for the original bytes by using the signature. The same could be done l 0x406AF3 by some const size, but regardless here it is:

```
(searches the virus body for the original bytes by using a signature, it also repairs th
reading original bytes directly to mapped file):

// --- snip of scanner code ------------------------------------------------
...(snip)...
unsigned char ctx_sig[15] = { 0x6A, 0x00, 0x6A, 0x05, 0xE8, 0x05, 0x00, 0x00, 0x00,
         0x90, 0x90, 0x90, 0x90, 0x90, 0x50 };
unsigned char ctx_fly[15];


    ReadProcessMemory(pi.hProcess, (void*)tc.Esp, &stack_v, 4, NULL);
    printf("[+] Virus request captured from 0x%.08x\n",stack_v);
    printf("[+] Scanning backwards to 0x%.08x\n",upa);


    while (1)
    {
        if (!ReadProcessMemory(pi.hProcess, (void*)stack_v, &ctx_fly,
                    sizeof(ctx_sig), NULL)) break;

        if (stack_v <= upa) break;
        found = 1;
        for (int ii=0; ii < sizeof(ctx_sig); ii++)
        {
            if (ctx_sig[ii] != ctx_fly[ii])
            {
                if (ctx_sig[ii] != 0x90)
                {
                    found = 0;
                    break;
                }
            }
        }

        if (found == 1)
        {
```

```
            printf("[+] Orginal bytes were found at 0x%.08x\n", stack_v + 9);

            printf("[!] Repairing the broken instruction.\n");
            ReadProcessMemory(pi.hProcess, (void*)(stack_v + 9) ,(void*) sv, 5, NULL);

            printf("[!] The file was disinfected!\n");
            getch();
            goto error_term;
        }

        stack_v--;
    }

    if (found == 0)
    {
        printf("[-] Error: no signature was found.\n");
        printf("[-] Disinfecting failed\n");
        goto error_term;
    }

...(snip)...
// --- snip of scanner code ---------------------------------------------
```

The full EPO heuristics scanner, together with Win32.CTX.Phage disinfector, is attached to the l
the paper. Here is a screenshot from that application, as shown in Figure 10.

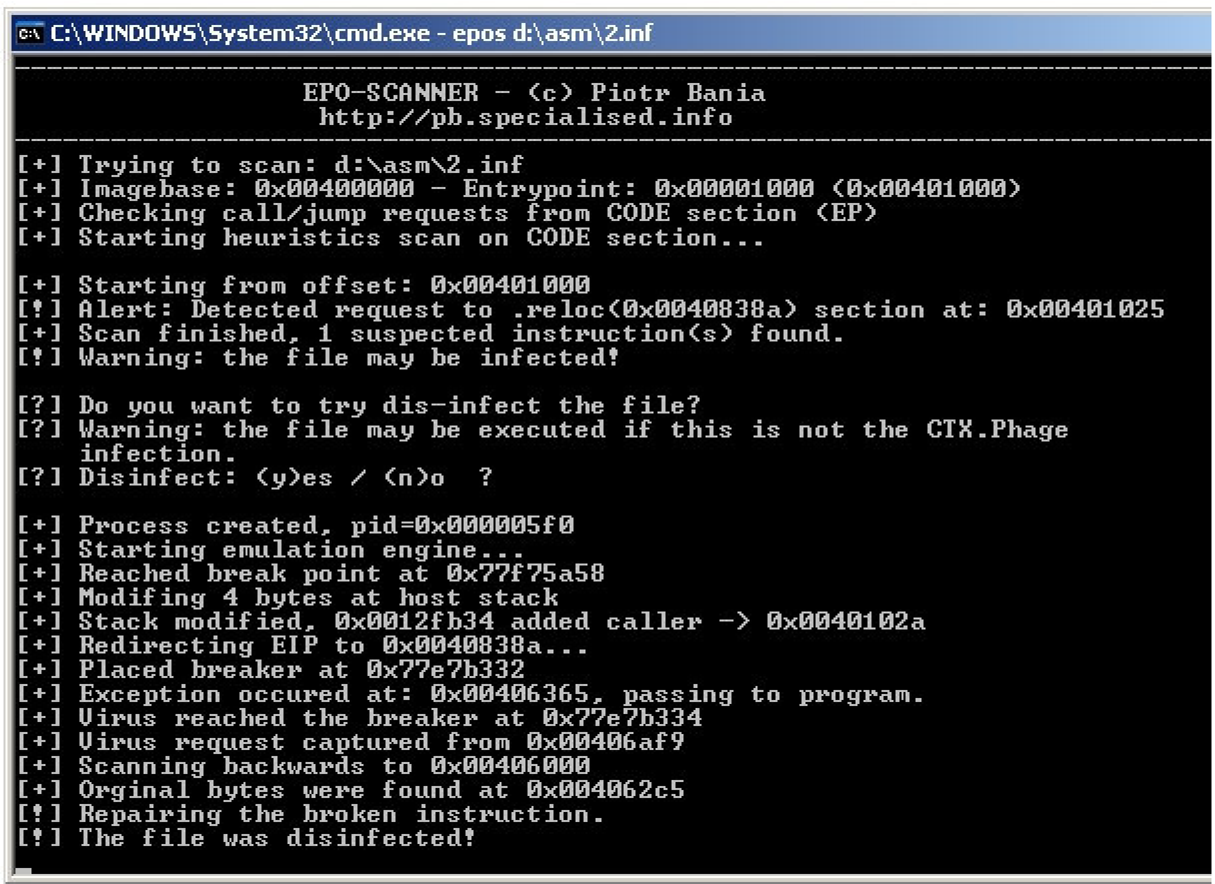


Figure 10. Screenshot of the EPO scanner.

### Curtains down – last words

I hope you have enjoyed this short article on EPO techniques. The disinfector discussed in this a
cancels virus injections, of course - the virus still resides in last section but fortunately it will ne

executed. However, this provides an opportunity for the reader to add some kind of virus "over
really an easy job and a good task to undertake.

If you have any comments don't hesitate to contact the author. The author would also like to tl
for moral support.

### Further Reading


1. http://securityresponse.symantec.com/avcenter/venc/data/w32.ctx.and.w32.cholera.htı Szor and Wason Han.
2. http://vx.netlux.org/29a/29a-4/29a-4.223 EPO by GriYo / 29a.
3. "The Art of Computer Virus Research and Defense" by Peter Szor


### About the author


Piotr Bania is an independent IT Security/Anti-Virus Researcher from Poland with over five yea
experience. He has discovered several highly critical security vulnerabilities in popular applicat
RealPlayer. More information can be found on his website.


### Code

Here is the full source code of the scanner and disinfector. If you have problems with formatti
source and precompiled binary is also available on SecurityFocus or through author's website.