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 de
disinfection and removal. Nowadays the EPO technique is used in many different ways, however
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 disinfec
tion 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 the
infectors use the most common way of doing this -- they simply change the entry-point of the i
application and make it point to the virus body. An example is shown below.

<table>
<thead>
<tr>
<th>Original EXE</th>
<th>Infected EXE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry-point: 0x1000 (.code section)</td>
<td>Entry-point: 0x6000 (.reloc section)</td>
</tr>
</tbody>
</table>

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 example
detects this type of infection:

```c
*(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 +
```
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 he 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 file section and searches for API calls generated by Borland or the Microsoft linker. When such code the virus checks that the destination address points somewhere inside the IMPORT section. If the code import call, Phage gets a random number which tells the virus to patch the current processed image then find next one. Figures 1, 2, 3, and 4 below show a few example schemas.
The above schemas show how the CTX.Phae 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 virus other hand, while using this EPO technique reduces the risk of virus execution, there are also some performance issues 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.Phae injections? First of all, the virus inserts a call instruction as
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 is possibility that we might find some "suspicious" call that hands in non-call instruction, for example:

```
68 332211E8  PUSH E8112233
```

As you can see, this is the push instruction, but the scanner finds the E8 byte and could consider it 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, here 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:

```c
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 instruction)
```

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

```
(seems 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);
```

http://www.securityfocus.com/print/infocus/1841

2005-07-07
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

                pSH->Name,pPE->OptionalHeader.ImageBase + temp_loc;
            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. Otherwise  
infected application may fault. The original instruction is saved somewhere inside the virus.

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 GetProcAddress API to obtain needed API addresses for their future execution. Lets try to set a breakpoint at GI (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.

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

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 GetProcAddress (nop), 0x90 (nop), 0xCC (int 3 – breakpoint) and takes over the EXCEPTION_BREAKPOINT on the "hooked" range:

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

// --- snip of scanner code ---------------------------------------------------------
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,
{ printf("[-] Error: cannot create process, error: %d\n",GetLastError());
goto error_di;
})
printf("[+] 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.ExceptionAddress);
            ContinueDebugEvent(de.dwProcessId,de.dwThreadId, DBREAK_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.ExceptionAddress);
    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),
                     PAGE_READWRITE, &oldp);
  }
}

oldp, &oldp);

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

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

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

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

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

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

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

![Figure 8. Checking the caller address.](image)

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

http://www.securityfocus.com/print/infocus/1841
2005-07-07
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 exist: inside the unencrypted body of virus, in other words, somewhere inside last section? Let's try to Figure 9.

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

```c
(unsigned char) ctx_sig[15] = { 0x6A, 0x00, 0x6A, 0x05, 0xE8, 0x05, 0x00, 0x00, 0x90, 0x90, 0x90, 0x90, 0x90, 0x50 };
unsigned char ctx_fly[15];

while (1)
{
   if (!ReadProcessMemory(pi.hProcess, (void*)tc.Esp, &stack_v, 4, 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)
   {
```

Figure 9. Scanning the virus.

http://www.securityfocus.com/print/infocus/1841

2005-07-07
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 disinfectior, is attached to the I the paper. Here is a screenshot from that application, as shown in Figure 10.

![Screenshot of EPO scanner](C:\WINDOWS\System32\cmd.exe - epos d:\asm\2.inf)

**Figure 10.** Screenshot of the EPO scanner.

**Curtains down – last words**

I hope you have enjoyed this short article on EPO techniques. The disinfectior discussed in this 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.ht](http://securityresponse.symantec.com/avcenter/venc/data/w32.ctx.and.w32.cholera.ht) by Szor and Wason Han.
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 year experience. He has discovered several highly critical security vulnerabilities in popular applicat RealPlayer. More information can be found on his [website](http://www.securityfocus.com/print/infocus/1841).

**Code**

Here is the full source code of the scanner and disinfector. If you have problems with formattin source and precompiled binary is also available on [SecurityFocus](http://www.securityfocus.com/print/infocus/1841) or through author's [website](http://www.securityfocus.com/print/infocus/1841).

---

**Privacy Statement**

Copyright 2005, SecurityFocus