

## ZeroAccess uses Self-Debugging

By Joshua Cannell

Published: 2013-07-24 · Archived: 2026-04-05 21:28:59 UTC

Debuggers—a tool traditionally used to find errors (called “bugs”) in code—are also used by security experts. In the field of [malware](#) analysis, debuggers are a vital tool used to reverse-engineer malware binaries, helping analysts to understand the purpose and functionality of malware when dynamic analysis isn’t enough.

Because they’re such a valuable tool, sometimes malware authors try to prevent analysts from using them. By employing various techniques in the code (known as “[anti-debugging](#)”), malware can successfully thwart junior analysts.

Recently I found an interesting anti-debugging technique I haven’t seen before. I discovered this technique while reversing a ZeroAccess Trojan (seems it’s always ZeroAccess lately, *right?*).

The technique employs various native Win32 APIs used for debugging a process. By using these APIs, the analyst cannot use their own debugger, since only one debugger can be attached to a process at a time.

To connect to the debugger at the API level, the Trojan uses *DbgUiConnectToDbg*. This API along with others used to communicate with the Windows Debugger all seem to be undocumented by Microsoft.

```
00401AE3
00401AE3 push    ebp
00401AE4 mov     ebp, esp
00401AE6 sub     esp, 64h
00401AE9 push    ebx
00401AEA push    esi
00401AEB push    edi
00401AEC call    ds:DbgUiConnectToDbg
00401AF2 test    eax, eax
00401AF4 jl     loc_401CB2
```

Next the Trojan creates a child process using the calling EXE (new-sirefef.exe). This was not surprising, as malware usually does this while unpacking. Allow me to explain.

```
00401AA5 push    eax                ; lpProcessInformation
00401AA6 lea    eax, [ebp+StartupInfo]
00401AA9 push    eax                ; lpStartupInfo
00401AAA xor     eax, eax
00401AAC push    eax                ; lpCurrentDirectory
00401AAD push    eax                ; lpEnvironment
00401AAE push    2000001h           ; dwCreationFlags
00401AB3 push    eax                ; bInheritHandles
00401AB4 push    eax                ; lpThreadAttributes
00401AB5 push    eax                ; lpProcessAttributes
00401AB6 push    [ebp+lpCommandLine] ; lpCommandLine
00401AB9 push    [ebp+lpApplicationName] ; lpApplicationName
00401ABC call    ds:CreateProcessW
```

Typically, a parent process creates a suspended child process using the calling EXE. Afterward, the parent will de-obfuscate some code and then place it in the child. Whenever this is complete, the parent makes a call to execute the child (usually with *ResumeThread*), which is now completely different from the calling EXE. And thus, while you have two processes that appear identical, they are completely different when viewed internally.

explorer.exe	16,360 K	23,800 K	1892	Windows Explorer	Microsoft Corporation
vmtoolsd.exe	8,292 K	12,952 K	1244	VMware Tools Core Service	VMware, Inc.
new-sirefef.exe	1,260 K	1,904 K	3488	Диспетчер синхронизации	Корпорация Майкрософт
new-sirefef.exe	1,396 K	3,016 K	964	Диспетчер синхронизации	Корпорация Майкрософт

This sample doesn't quite work this way. Under the creation flags parameter for the *CreateProcess* function, the *CREATE\_SUSPENDED* flag was not being used, but instead the *DEBUG\_PROCESS* flag. There was also another used, called *CREATE\_PRESERVE\_CODE\_AUTHZ\_LEVEL* (Note: for a list of process creation flags, click [here](#)).

```
ModuleFileName = "C:\\Documents and Settings\\Administrator\\Desktop\\new-sirefef.exe"
CommandLine = "\"C:\\Documents and Settings\\Administrator\\Desktop\\new-sirefef.exe\"\""
pProcessSecurity = NULL
pThreadSecurity = NULL
InheritHandles = FALSE
CreationFlags = DEBUG_PROCESS|20000000
pEnvironment = NULL
CurrentDir = NULL
pStartupInfo = 0012FEDC
lpProcessInfo = 0012FF20
```

Now both the parent and child process are being debugged, which means we can't attach an additional debugger to either. This complicates matters as the debugger is the primary tool we use to step through code.

However, we can still observe what's happening statically using our IDA dump. The parent process appears to handle debug event codes and performs an action for each event (for a list of all codes, see [here](#)). After an event has been processed the Trojan continues debugging and receives another event using *DbgUiContinue*.

```
00401C80
00401C80 CONTINUE_TO_NEXT_EVENT:
00401C80 push [ebp+lpApplicationName]
00401C83 mov esi, [ebp+var_4]
00401C86 lea eax, [esi+4]
00401C89 push eax
00401C8A call ds:DbgUiContinue
00401C90 cmp [ebp+lpCommandLine], ebx
00401C93 jz short loc_401CA5
```

When an *EXCEPTION\_DEBUG\_EVENT* code is received, the Trojan enters a function that decrypts a PE DLL file to the heap. The new PE is then placed into the memory space of the child process.

Address	Hex dump	ASCII
00163D88	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ□.□...□...ÿÿ..
00163D98	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	,.....@.....
00163DA8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00163DB8	00 00 00 00 00 00 00 00 00 00 00 00 F0 00 00 00	.....ð...
00163DC8	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	□□°□. '.Í!_□LÍ!Th
00163DD8	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
00163DE8	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00163DF8	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....\$.....
00163E08	7A EA 5C 89 3E 8B 32 DA 3E 8B 32 DA 3E 8B 32 DA	zê\%><2Ú><2Ú><2Ú
00163E18	81 84 52 DA 3D 8B 32 DA 3E 8B 33 DA 48 8B 32 DA	□,RÚ=<2Ú><3ÚH<2Ú
00163E28	FD 84 6F DA 33 8B 32 DA 19 4D 4F DA 3C 8B 32 DA	ÿ,óÚ3<2Ú□□MOÚ<<2Ú

The new PE file is actually the final unpacked version of the rootkit. We can dump the memory from here and load it into IDA to perform some static analysis. Looks like we have some websites in plain-text the Trojan is going to contact, possibly to locate the infected user (geiop\_country\_code).

```

[s] .rdat... 00000007 C fp.exe
[s] .rdat... 0000009C C GET /count.php?page=%u&style=LED_g&nbdigits=9 HTTP/1.1\r\nHost: www.e-zeeinternet.com\r\n
[s] .rdat... 00000047 C GET /app/geiop.js HTTP/1.0\r\nHost: j.maxmind.com\r\nConnection: close\r\n\r\n
[s] .rdat... 00000014 C geiop_country_code
[s] .rdat... 0000000E C j.maxmind.com
[s] .rdat... 00000010 C ShellExecuteExW

```

This is just another example of how malware authors attempt to prevent reverse-engineering of their code with anti-debugging. In this example, however, the ZeroAccess Trojan does not allow the analyst to use their own debugger by connecting to the Windows Debugger itself. All in all I think it's a very interesting technique, and we're sure to see more of it in the future.

---

[Joshua Cannell](#) is a Malware Intelligence Analyst at Malwarebytes where he performs research and in-depth analysis on current malware threats. He has over 5 years of experience working with US defense intelligence agencies where he analyzed malware and developed defense strategies through reverse engineering techniques. His articles on the *Unpacked* blog feature the latest news in malware as well as full-length technical analysis. Follow him on Twitter [@joshcannell](#)

### About the author

Gathers threat intelligence and reverse engineers malware like a boss.