

Advanced Windows 2000 Rootkit Detection (Execution Path Analysis)

Jan Krzysztof Rutkowski
jkrutkowski@elka.pw.edu.pl

July 2003

Abstract. In the article a new approach to detection of kernel- and user-mode rootkits has been described. Presented technique exploits the processor stepping mode to measure the number of instructions executed in system kernel and DLLs, in order to detect additional instructions inserted by malicious code, like rootkits, backdoors, etc... Implementation of simple detection utility for Windows 2000, which makes use of this technique, is also discussed.

Background

One of the most important problem in computer security is, how to check if given machine has been compromised or not. This is very difficult task, because of two factors:

- attacker can exploit unknown bug in the software to get into the system
- after break in, attacker can install advanced rootkits and backdoors in order to stay invisible (i.e. hide processes, communication channel, files, etc...).

In this paper we will concentrate on rootkit detection in Windows 2000 systems.

Problems with traditional rootkit detection

Traditional rootkit detection programs (which we can mostly encounter on UNIX systems) are trying either to detect only known rootkits (which makes them similar to anti-virus software) or perform some kind of kernel memory scanning.

For example, some tools has been created for the Linux OS , which scans kernel *syscall table*. This is obviously not sufficient, because many rootkits, which does not touch *syscall table*, has been created. Similar rootkits can be developed for Windows 2000.

One can then think, that such detectors should also scan kernel code area, so we will have something similar to famous Tripwire, but working in kernel mode. Surprisingly this is still not enough, because on many operating systems we can write kernel rootkit, which does not change SST (*syscall table*) nor the code. In system kernels there are many function pointers which can be hooked (see [2] for example)...

Memory scanning approach, in the author's opinion, will never lead to complete rootkit detection, mostly because nobody can point which memory areas should be monitored (because their change means something wrong happened) and which should not (because they are dynamic data, changing hundred times every second).

How to detect intruder in our system then?

Rootkit classification

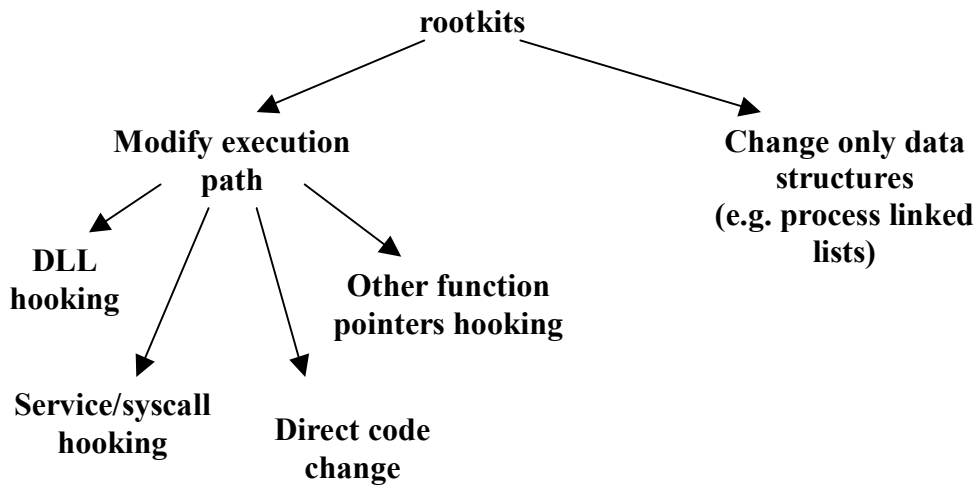


Figure 1. Rootkit classification.

First we should divide techniques exploited by rootkits into two categories (see figure 1): these which only modify some kernel data structures in order to hide something (like active processes) and these which achieve the same result by modifying some kernel execution paths (for e.g. hooks some kernel functions responsible for enumerating active processes).

Rootkits which change kernel data structures

There are not many rootkits which makes use of this technique. Interesting example within this category is *fu* rootkit (see [3]), which hides processes by just unlinking process objects from the *PsActiveProcessList* linked list in the kernel. Hidden process is invisible to *ZwQuerySystemInformation* function (for e.g.) which is used by tools like Task Manager to get list of processes running in the system.

On the other hand, hidden process is able to execute (i.e. it gets CPU time) since different lists are used by the dispatcher (windows scheduler). Actually Windows dispatcher is using three¹ important data structures to manage thread scheduling, they are:

- *pKiDispatcherReadyListHead*, which is actually a table of 32 linked list heads (one list for each priority) grouping threads in 'ready' state.
- *pKiWaitInListHead*
- *pKiWaitOutListHead*

The last two are heads of two different lists which holds threads being in state 'wait' (like sleeping on some object). There is subtle difference between them, but this is not important for us.

¹ Author believes that this three lists cover all threads in the system. Some experiments has proved this assumption, however this should be better researched.

From the above we can see the easy way to detect hidden processes. We should just use the utility (kernel module) which will read the internal dispatcher thread lists, instead of the *PsActiveProcessList*. Such utility has been developed and proved to detect all hidden, by *fu* rootkit, processes.

We should see now that detection of similar kind of rootkits is relatively easy: we should just reach the most internal kernel data structures, read them (of course we need own kernel module to get to the kernel data space), and then compare the results with ones returned by some “official” API functions.

Please note, that it is rather impossible to remove threads of the hidden process from the just mentioned dispatcher internal lists, because then, our hidden process, will not get any CPU time for execution.

Rootkits which change Execution Path

These are the most common rootkits. They achieve their goals by altering or adding some extra instructions to the kernel or system DLLs. The problem is, that we do not know where or what rootkit will change. It can hook some DLL functions or System Service Table, change body of some kernel functions or just change strange function pointers in the kernel...

The problem is that kernel is so big structure, that we do not actually know what areas of its memory should be checked in order to detect intruders. We cannot check too much, since there are lots of fast changing data in kernel (processes are running, packets from the network are coming, etc...) and we do not want to get false positives every second!

Execution path analysis

The main idea in EPA is to notice the simple fact: if the attacker has compromised the system, and she is trying to hide something by changing some execution path, then system will be executing extra instructions, during some typical system and library calls.

For example, if she patched *ZwQueryDirectoryFile()* and *ZwQuerySystemInformation()*, in order to hide files and processes then, the number of instructions which are executed during these system services will be greater, comparing to the clear system. It does not matter if attacker has hooked some entries in System Service Table (SST) or put some *jmp*'s into code, or do whatever else. More instructions are executed because rootkit must perform its task (remove some elements form the returned lists of files or processes in this example).

The Windows 2000 kernel is a complex program however. We suspect, that even in the clear system, the number of instructions executed during some system calls will not be the same. However, as we will see later, we can deal with this situation using simple statistics. But first, we need a mechanism for instruction counting...

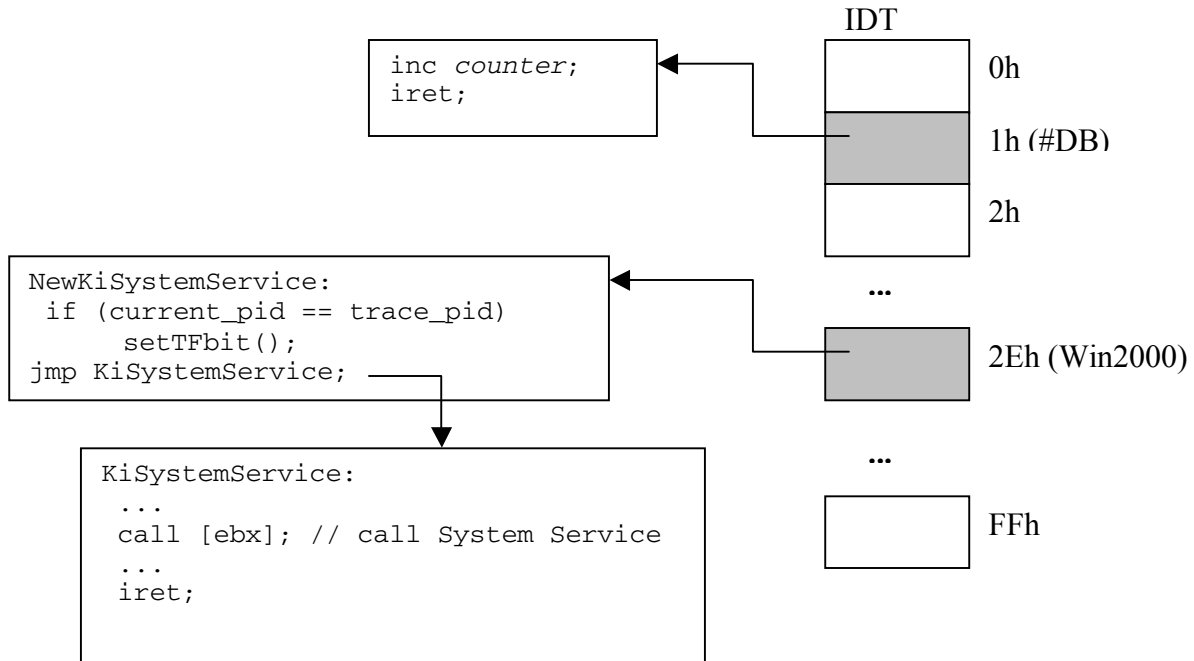


Figure 2. Simplified diagram of what is hooked by instruction counter.

Implementation of instruction counter

To implement instruction counting, we can use a nice feature of Intel processors, the so called single stepping mode. When the processor is working in this mode, it will generate debug exception (#DB) after every instruction which was executed. To put processor into this mode, we have to set TF bit in the EFLAGS register (see [6] for details).

Processor clears TF bit when executing `int` instruction and this instruction is causing privilege change. It means that if we want to count the instructions executed in kernel mode, we have to set TF bit at the beginning of interrupt handler. Because we would like to measure the number of instructions executed during some system services, it is convenient to hook interrupt vector `0x2e`, which is Windows 2000 system service gate.

However, because some rootkits are user-mode based (see [4]), we should also be able to count the instructions executed in `ring3` before, and after, kernel service. This is simple, we just have to set TF bit in user mode and we do not have to bother to set it again after return from kernel-mode, because processor will restore this bit automatically.

The counting mechanism has been implemented as a kernel driver. After loading, it hooks IDT `0x1` and `0x2e` entries as showed on figure 2. Of course the module must communicate with userland to allow testing process getting the results. In current implementation kernel driver hooks one system call which will be used as a interface to the driver. User mode process can then start and stop the counting process by calling this specific system call.

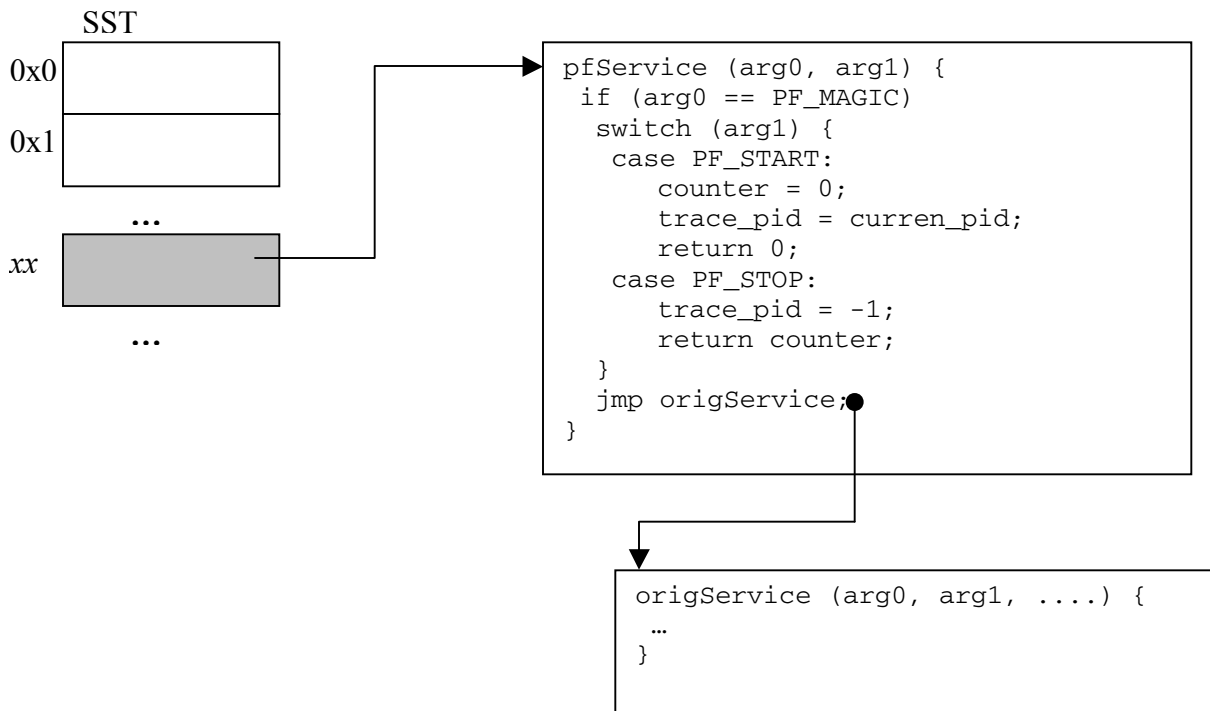


Figure 3. Kernel module communicate with user mode testing process through one of the system service.

The tests

We can use the mechanism described in the previous paragraph to measure the number of instructions executed during arbitrary system services.

For example, to detect that someone is trying to hide some files or directories, we can issue a simple test:

```

pfStart();
    FindFirstFile("C:\\WINNT\\system32\\drivers",
                &FindFileData);
int res = pfStop();

```

We expect that if attacker has installed rootkit which hides arbitrary files, then we will count more instructions compared to the uninfected system.

If we repeat the tests few hundred times and calculate the average value, we can see that tests are very non deterministic. It should not be surprise, considering such complex system as Windows 2000. Unfortunately such behavior is unacceptable for our detection purposes. However, if we collect our test samples and create histogram we can observe that all tests are characterized by a histogram with one big peak. This peak, as it can be seen on the pictures 4 and 5, remains at the same position, even though the system can be heavy loaded. In this case, the histogram has been created for the *FindFirstFile()* function.

This surprising behavior of system functions is hard to explain formally. Intuitively we expect that, because one system service is called few hundred times in a loop, so that all system buffers, caches etc., connected with serving this special service, will be finally filled with some fixed values.

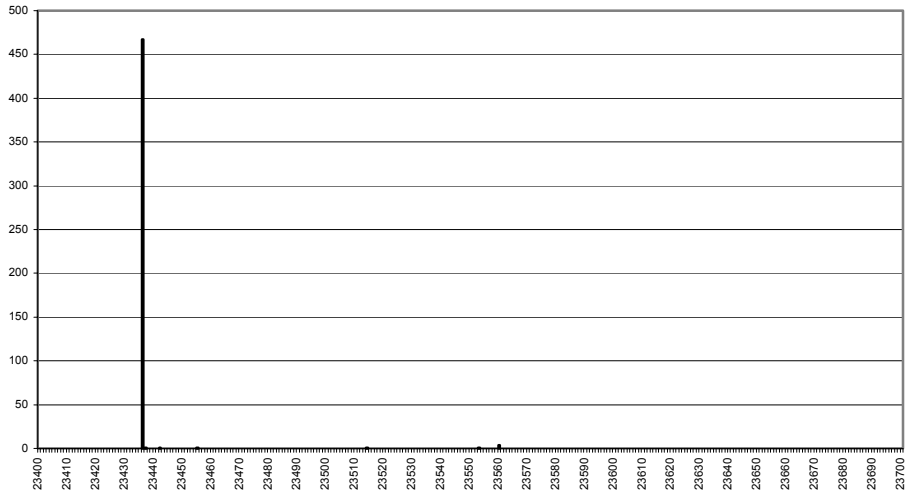


Figure 4. Histogram for the *FindFirstFile* test, system under light load. The number of instructions executed both in kernel and user mode has been counted, while testing process was asking system to return the first file form `\WINNT\System32\drivers` directory. This directory has been chosen because its contents shouldn't change.

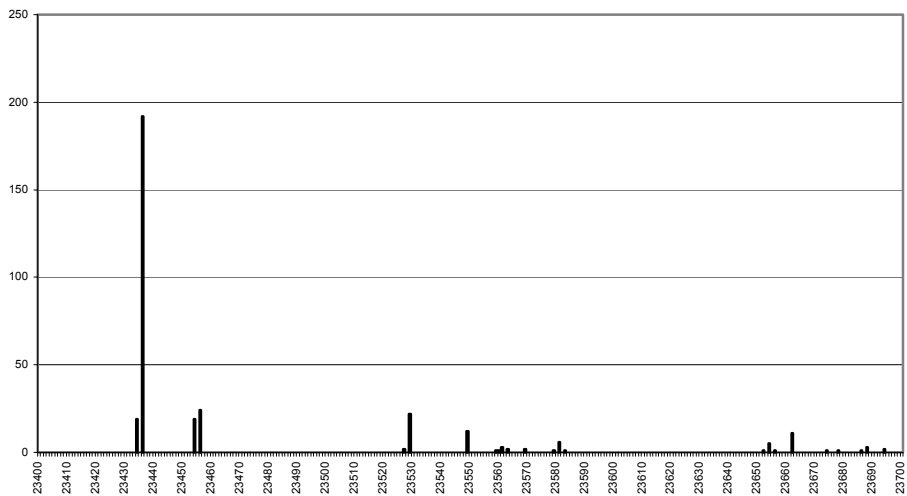


Figure 5. Histogram for the *FindFirstFile* test, system under high load. Notice that main peak remains at the same position.

Imagine now, that somebody has installed a rootkit which hides files. Then if we repeat the tests and draw a corresponding histogram we will notice that peak has been moved

towards right side. This is because rootkit must perform additional operations, in this case, remove the hidden files from the returned list.

In current implementation only a few tests has been used. They check such typical services like reading directory contents, enumerating processes, enumerating registry keys and reading from the network socket.

These tests are sufficient to detect such rootkits like famous *NTRootkit* (see [1]), or recently very popular *Hacker Defender* (see [4]) including its clever network backdoor and of course many more. However some new tests should be added to catch other smart network backdoors.

False positives and execution path tracing

Although peek detection technique allows us to deal with no deterministic nature of tests quite well, sometimes we can observe small differences in peak position. The deviation is about few instructions (generally not more then twenty).

Sometimes this can be a serious problem, since we must decide weather these few additional instructions means system compromise or just a noise.

To solve this dilemma, Execution Path Recording should be used. In contrast to pure EPA, this time, debug handler records the whole path which was executed (addresses and instructions at this addresses). Then we find the most common path (with the biggest peak) and make a diff of these two most common paths (the current one and the saved one).

We should analyze this extra instructions with a good disassembler and decide weather they are suspicious or not. Example diff between two tests one is shown of figure 6.

In current implementation only the addresses are dumped to the trace files. In future release the traces will probably be saved as PE files, which will enable the use of some 3rd party disassemblers (like IDA) .

```

*** RegEnumKey-clear.trace
--- RegEnumKey-current.trace
*****
*** 273,278 ****
--- 273,281 ----
 0x80416f60
 0x80416f63
 0x80416f65
+ 0x80416f67
+ 0x80416f6a
+ 0x80416f6d
 0x80416f74
 0x80416f76
 0x80416f77
*****
*** 1002,1007 ****
--- 1005,1013 ----
 0x80416f60
 0x80416f63
 0x80416f65
+ 0x80416f67
+ 0x80416f6a
+ 0x80416f6d
 0x80416f74
 0x80416f76
 0x80416f77

```

Figure 6. Differences between two trace dumps (only addresses). We see that 3 additional instruction has been executed two times (after 275th and 1007th instruction).

Detection of ‘offset-in-the-code’ changes

Imagine a rootkit which works similarly to *fu* rootkit ([3]) mentioned above, but instead of removing objects from *PsActiveProcessList* its removes the thread objects from the internal dispatcher data structures. We said that its not possible, since our hidden process will not get any CPU time for execution...

However we can imagine that rootkit will also change to scheduler code in the fashion it will use different offsets (within thread objects) to maintain a list. In other words we change scheduler code to use different (“shadow”) list. But only the scheduler will be changed to use this “new” list, everybody else (including IDS tools, like those mentioned above which can read internal kernel structures) will be using the old list... See figure 7.

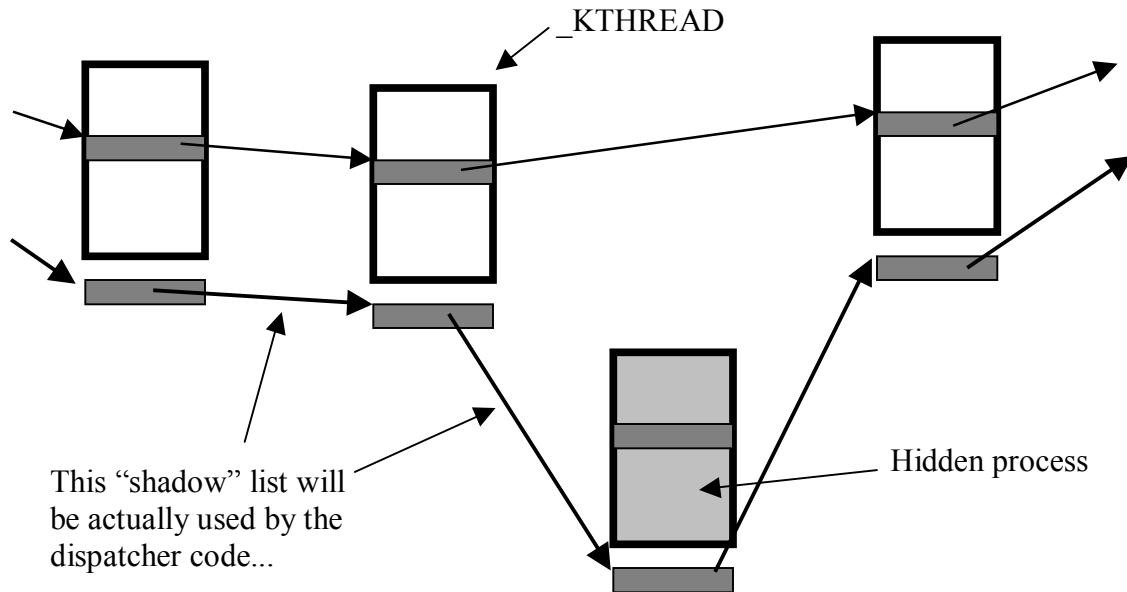


Figure 7. Example of „shadow” threads list (dispatcher code modification is needed, actually only offsets should be changed).

Although this technique won't change the number of executed instructions, we should be able to detect it, provided we will have Execution Path Recording fully implemented (i.e. we will detect that two traces differ, at positions where the offsets have been „patched”).

This has not been implemented yet, but should be relatively easy to do.

EPA bypassing and defense

We can think of few ways of cheating presented detector. Let's divide these attacks into two groups:

- cheating specific tool (attacker knows exactly the binary of kernel driver and tester program)
- more general attacks against the EPA technique.

First, we will consider general ways of cheating detector and how to defend against them. Then we will discuss vulnerabilities of specific tool and how to avoid them by using polymorphism.

General attacks and defense

First of all malicious software can hook IDT entry no 1, which contains the address of debug handler. Then instructions won't be counted. After rootkit finished its job, it should unhook IDT debug entry. As a result testing process won't count the extra rootkit instructions.

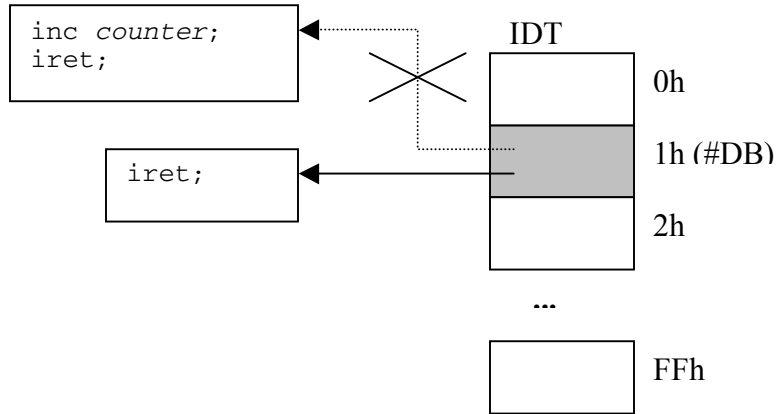


Figure 8. Simple attack against EPA.

To defend against such attack, we can make use of Intel debug registers. We can use DR0 and DR1 registers to protect whole IDT 1 entry (which is 8 bytes long) against writing access. In addition we would like to protect this entry against reading, to not allow hooking debug handler by just inserting `jmp` instructions somewhere at the beginning of the debug handler. In other words, we do not want the rootkit to be able to find the address of debug handler.

However we can not simply protect IDT #1 from reading (by setting appropriate flags in DR7) because the system will bluescreen, as experience has showed. But there is a simple solution, which involves adding an extra layer. The solution is depicted on figure 9.

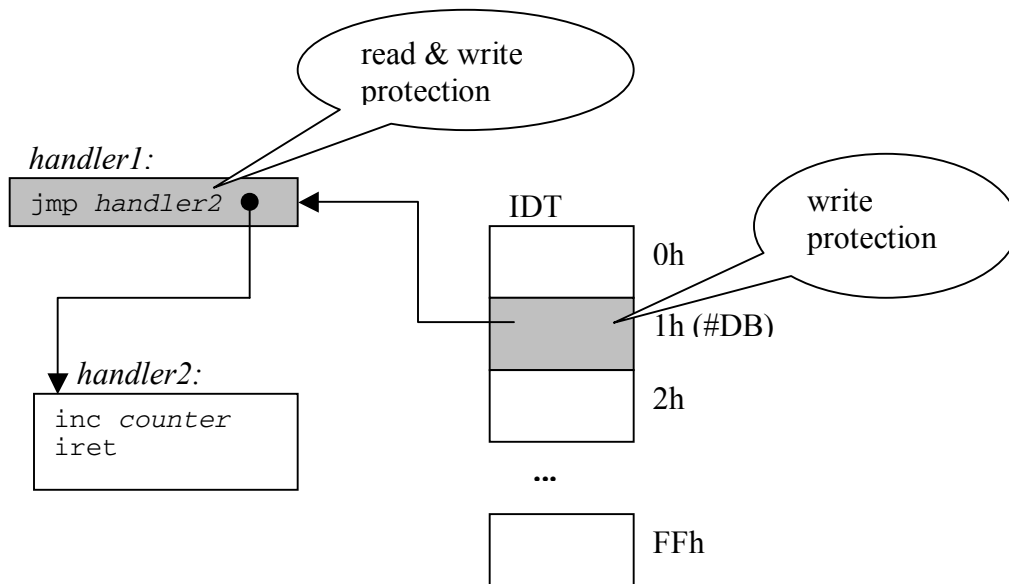


Figure 9. Hardware protection against #DB hooking.

Another attack involves writing such rootkit, that clears TF bit at the beginning of its body, then does whatever it wants (like removing some processes from the list), and set TF bit again. Tester process again will report only few more instructions executed, which would look like a clear system.

Alternatively rootkit can check TF bit in EFLAGS and, when it finds out that it is being traced, do not execute its malicious instructions. Such behavior does not betray rootkit, because rootkit detects TF bit only when its instructions are executed in the context of testing process.

We can defend against such attacks. It should be noted that our debug handler is called before any instruction to be executed.

Simple defense can be as follow. If debug handler detects that previous executed instruction was `pushf` (which stores EFLAFS register onto the stack) then it should perform the following operation:

```
and [esp], 0xfffffeff;
```

which will simulate that TF bit is not set in the EFLAGS. Similarly debug handler should check if the next instruction to be executed is `popf` (which loads EFLAGS with the word found at the top of the stack). And if it is, debug handler should execute the following code:

```
or [esp], 0x100;
```

which sets TF bit, so that if rootkit had tried to clear it would not succeed.

This line of defense seems reasonable, but is not sufficient. Rootkit can do the following in order to detect that somebody is tracing it:

```
setTFbit();
if (checkTFbit() != 1) {
    // we are traced!
}
```

We see that we should modify our anti-detection mechanism, so that it will remember weather TF bit has been set by the traced process or not (by adding an extra variable called *TFbitset* for example).

It should also be noted, that `popf`/`pushf` are not the only instructions which are used to access EFLAGS register. Similar functionality we can also observe in the `iret`/`int` and few others instructions², so that similar steps should be taken.

EFLAGS protection has not been implemented yet, partly because we should be able to detect such tricks with `pushf`/`popf` instructions, by analyzing diffs of execution traces. However, this should be implemented in future releases.

Attacks against specific tool

If the attacker knows everything about the tool, which implements presented technique, she can cheat it in many ways. For example, she knows (or can find it very easily, using pattern searching for example) where, in kernel area, is stored *counter* variable, which is incremented by debug handler.

Such attacks are expected to be implemented by rootkit authors, only when a specific tool will become quite popular. If we had got many tools implementing EPA (or many versions of one tool), such attack won't be profitable for the attacker.

However, we would like to be also resistant for such implementation based attacks. Probably the only way of doing it, is strong polymorphic code generator. After administrator has downloaded such tool into his system, during installation phase, a unique kernel driver and testing program should be generated.

Polymorphic generator has not yet been implemented.

Related Work

As it has been said at the beginning, author is not aware of any other general approach to rootkit detection, which is not based only on memory scanning.

It should be noted that presented technique is not very OS specific, and author has also implemented a similar detection utility on Linux system. See [5] for more details and very simple proof of concept code for Linux 2.4.

References

- [1] Greg Hoglund, et al, *ROOTKIT home*, <telnet://rootkit.com>,
- [2] palmers, *Sub proc_root Quando Sumus*, Phrack Magazine, issue 58, 2001.
- [3] fuzen_op, *fu rootkit*, <telnet://rootkit.com>,
- [4] Holy Father, *Hacker Defender Home*, <http://rootkit.host.sk>,
- [5] Jan Rutkowski, *Execution path analysis*, Phrack Magazine, issue 59, 2002.
- [6] IA-32 Intel Architecture Software Developer's Manual, voll1-3.

² The list of all instructions which interact with EFLAGS register can be obtained from [6].