Concepts for the Stealth Windows Rootkit
(The Chameleon Project)

Joanna Rutkowska
joanna@mailsnare.net

September 2003

Purpose
Many people do not realize the real danger from rootkit technology. One reason for this is probably that publicly available rootkits for Windows OS are relatively easy to detect by conventional methods (i.e. memory scanning based). However, we can imagine some techniques of rootkit implementation, which will be undetectable by these methods, even if the rootkit concept will be publicly available… In order to convince people that traditional rootkit detection is insufficient it would be desirable to have a working rootkit implementing such sophisticated technology. Besides it would be fun.

Assumptions
The rootkit, which we are going to discuss, should provide only two features:
- Process hiding
- Network Covert Channel

Rootkit should be undetectable by memory scanning-based methods, with the assumption, that everybody knows the technique it exploits. In this way it should be similar to modern cryptographic algorithms, where the strength of the cipher is not based on the obscurity of the encrypting method but on the secret key.

Rootkit should work on Windows 2000 and 2003 systems. Server systems are of the primary concern, since rootkit is not going to employ any mechanisms to survive system restart, because such actions cannot be made stealth enough (*vide* Forensic Analysis).

Traditional process hiding techniques
There are many techniques for hiding processes in the system. Lets take a brief look at them and discuss their weaknesses:

- **DLL infection** (file replacement) – new DLL (NTDLL.DLL for e.g.) is provided with some functions patched (like *ZwQuerySystemInformation*). Easily detectable by file system integrity checkers (provided they are not using DLL, but direct *int 2eh* calls)[1].

- **API hooking** – almost the same as above, but no file replacement is needed. Infection is done by means of *OpenProcess()*, *WriteProcessMemory()*, etc… Example rootkit of this kind is HackerDefender [1]. There is similar problem here to the above, although the tool for checking integrity of process address space should be little more sophisticated [2]. There is also another method of disclosing all hidden process by this and above methods. Administrator can use a

[1] It would be interested if someone will check such commercial products like Tripwire or Intact, if they are using NTDLL functions to read files or not.

small program which will call *ZwQuerySystemInfo()*, not through the NTDLL, but directly through *int 2eh*. This is the problem of all userland rootkits, which tries to hide some objects. They can always be uncovered in this way.

- **"Thread injection"**– in this technique there is no additional process creation. Rootkit chooses appropriate "legal" process, and inserts a new thread into this process. This thread is doing something useful for the attacker. The problem here is the flexibility. Although some task could be implemented this way, it would be probably impossible to have a running hidden console and the possibility to start new different, arbitrary programs from this console. It is of little chances that some, even medium-complicated programs, could be made running this way.

- **Kernel service hooking.** This is a classic approach, which involves hooking some well known kernel services and cheating about the results they return (see [3] for some code examples). It can be implemented in several variants:

    1. **Service Table hooking:** probably the most popular approach, which is also easy detectable by comparing ST with the copy from the "clear" system.
    2. **IDT hooking:** similar to the above, similar disadvantages.
    3. **Raw kernel code change:** instead of changing pointers to code, we just changing the code itself, by inserting for e.g. some 'jmp' instructions.
    4. **"Strange" pointers change:** we are changing here some uncommon code pointer (in contrast to ST or IDT), in order to change the execution path of some kernel services. There are many such pointers, for example *pServiceDescriptorTable* in _KTHREAD object.

  Rootkit implementing techniques 1-3, are relatively easy to detect by calculating hashes on kernel memory, see again [2] for some details. The 4th approach is little more problematic, since its probably impossible to spot all the valuable (from the attacker's point of view) pointers in the kernel, which should be checked, and their change means system compromise, not the daily kernel activity.

  We will be back to 4th method weaknesses in a while.

- **Kernel data manipulation.** Instead of changing kernel code (or execution path), the idea here is to just unlink the process from the *PsActiveProcessLinkHead* list. This list is not used by the scheduler code, so after unlink, process (or its threads actually) still gets the CPU for execution, while the process object is hidden. To detect hidden processes in this way, we have to read the lists which the scheduler actually uses.

Scheduler structures

Scheduling in Windows is thread-based, so there are lists which group thread in different state (running, sleeping, etc). If rootkit unlinked threads of the hidden process from this lists too, then those threads wouldn't get any CPU time, make such rootkit useless. A simple tool, *klister*, for reading such internal scheduler lists has been released [5]. It should be noted that this tool is also capable of detecting hidden processes by all the previously described methods (including 4th one from the previous paragraph).

Author has currently identified three lists used by the scheduler to maintain the threads, they are[2]:

- *KiDispatcherReadyListHead* (which is actually a table of 32 lists, for every priority each),
- *KiWaitInListHead* and *KiWaitOutListHead*.

The last two lists are very similar in their nature and they group the threads which are in a waiting state[3]. Although the hooks of the lists are different, all the lists are implemented by the *WaitListEntry* field in the KTHREAD object (offset `+0x5c`).

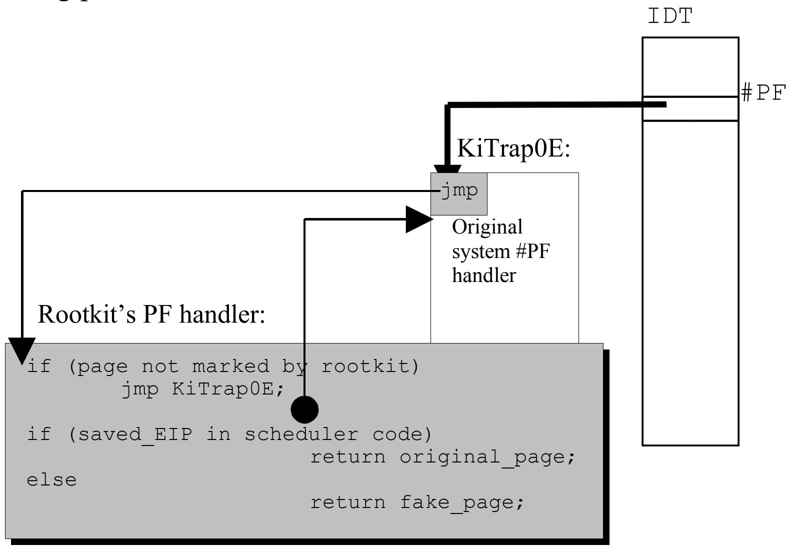
"Innocent threads" concept

It should be obvious now, that we need new, more stealth, technique for hiding processes in the system. What we want, is that all threads from the hidden process, disappear from all this scheduler lists. But we can not remove threads from these lists, since then, the process won't get any CPU time…

The idea is to make the threads of the hidden process look like threads which belong to some system process like *winlogon.exe* for example. Of course we should first unlink the hidden process object from *PsActiveProcessLinkHead*, in exactly the same way as *fu* rootkit does it (see [4]).

We should then cheat when somebody access the page where such thread object is stored. However, we should not cheat when the access to this page comes from the scheduler code. This means that Page Fault (#PF) handler should be hooked. PF handler can be hooked either by changing entry in IDT table either by replacing the first few instructions of the original system PF handler (*KiTrap0E*). Since IDT hooking is easy detectable, it would be better to use the second approach. This can be also detected by utility similar to Tripwire, but running in kernel memory space. We will back to this issue later and discuss how to make such changes undetectable.

We have the following picture then:



```
if (page not marked by rootkit)
      jmp KiTrap0E;

if (saved_EIP in scheduler code)
                        return original_page;
else
                        return fake_page;
```

We also need to mark all the pages, which we are going to cheat about, so that access to those pages will cause #PF exception. This can (probably) be done by clearing the *present* bit in the appropriate page table entry. Rootkit's #PF handler should determine then, weather the access come from the

[2] The symbols are not exported, but addresses cab be obtained from the debug symbol files.
[3] Author didn't investigate what is the real difference between these two lists.

scheduler code or not, and return either the original page contents either the fake one (for e.g. with the contents of fake thread object which looks like thread from *winlogon.exe*).

The smart way of deciding weather the access came from scheduler or not should be researched[4]. The first idea here is just to check if EIP[5] belongs to the address space occupied by the *ntoskrnl.exe* module (its code section to be more precise).

Rootkit should also be able to decide weather the page with *present* bit cleared was marked by rootkit or by OS. This should be probably best solved by storing some magic tag in the page contents (not page table entry).

We should also modify *NtCreateThread* in such a way that if a hidden process will create new thread, the page where its object is held will be marked by clearing the *present* bit in the page table entry. Thanks to this, when we start new process, we do not have to worry how many threads (and when) it will create.

Fake threads

Ok, so how does the fake thread object should look like? We have much flexibility here, because we know that whatever we put here, OS won't use it for anything important. That's why we can probably just chose some arbitrary thread object from some popular system process, like *svchost.exe*, and copy its contents as a new fake thread object. However, we should change some fields, so there were not two exactly the same threads in the systems (this could be easy detectable). It should be further investigated what files should be changed to make such thread looking like just another thread of some system process.

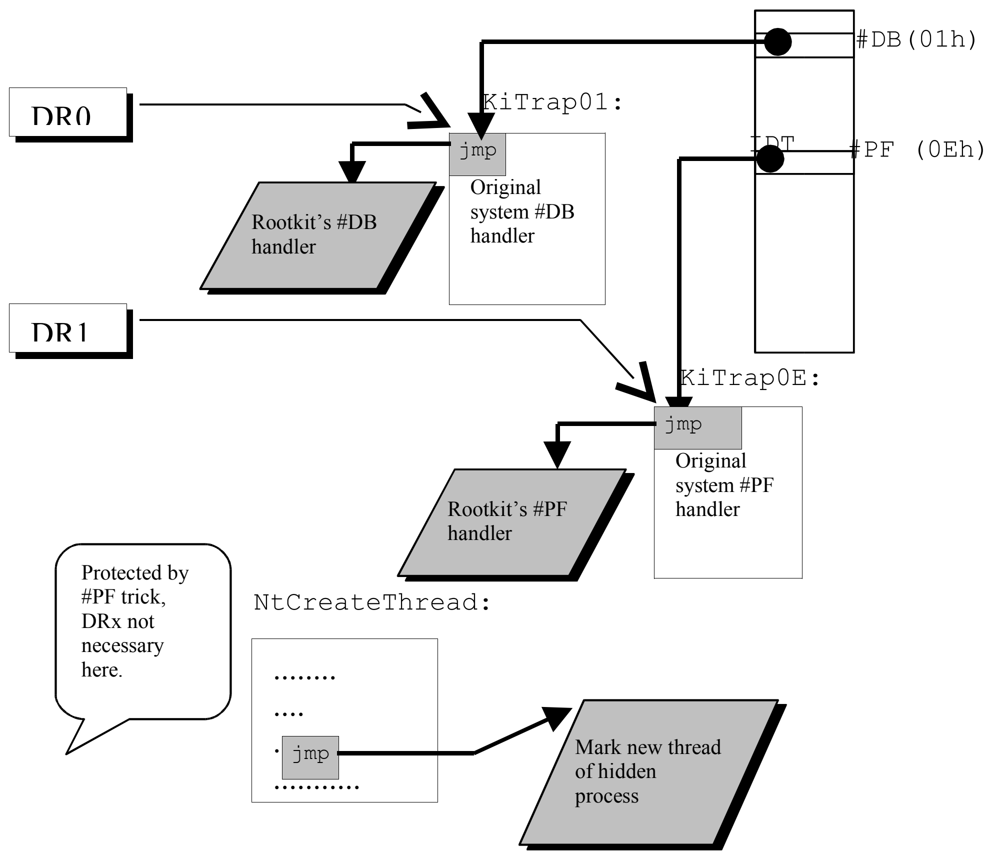
Hiding the code changes

The approach presented up to this point has one weak point. The administrator can detect that someone has changed the first few bytes of the original Windows #PF handler (*KiTrap0E*).

We cannot make the page in the memory where the original PF handler resides inaccessible (by clearing the *Present* bit) because of the obvious recursion problem here.

Probably the only way to solve this problem is to use the Debug Resisters of the Intel Processors. We can set the DR0[6] to point at the beginning of *KiTrap0E*, where we put "jmp" instructions to jump into rootkit PF handler. We should also set some bits in DR7 (aka Debug Control Register) to cause #DB exception when read access to the address pointed by appropriate DRx register. Rootkit should then provided its own #DB handler, again by inserting some "jmp" instructions at the beginning of the original system #DB handler – *KiTrap01*. This modified bytes should also be protected by Debug resister (DR1). No recursion should occur here, because we set the breakpoint here for data read/write, and not on instruction fetches. The situation is depictured below.

[4] It should be also noted, that not only scheduler can access the thread objects, that's why all the OS modules that make use of them should be identified (for e.g. hal.sys, ntoskrenl.exe, etc…) and all access from their code sections should see the original, not faked, objects.
[5] The one saved by during exception.
[6] It should be noted that one DRx register can protect up to four bytes. So, care should be take to modify as little bytes as possible, since we have only four debug address registers (DR0, DR1, DR2 and DR3).


#DB(01h)
DR0
KiTrap01:
jmp
IDT
#PF (0Eh)
Rootkit's #DB handler
Original system #DB handler
DR1
KiTrap0E:
jmp
Rootkit's #PF handler
Original system #PF handler
Protected by #PF trick, DRx not necessary here.
NtCreateThread:
jmp
Mark new thread of hidden process

In addition, rootkit should set GD flag in DR7 in order to protect access to DRx registers. When GD is set, any access to DRx registers causes #DB to be generated, and BD flag in DR6 to be set. DB handler can detect such tries and either ignores them, either make some action, like restarting the machine or something more sophisticated. See the next paragraph for more details about this.

Protecting DRx registers

Using Debug Registers together with #PF hooking to protect arbitrary memory areas should solve the problems of detecting rootkit by memory scanning[7]. However, the usage of DRx registers opens one more way for detecting our rootkit. Please note that we are using two out of four available Debug Address Registers. We can imagine then a rootkit detector which tries to make use of the all four DRx registers (by setting some test breakpoints and check weather they are hit or not). This would either break our rootkit protection scheme (if rootkit did not set GD flag in DR7) either detector will realize that test breakpoint has not been hit.

As it was said above, an extremely easy way to get around this is to execute `int3` instruction when detected BD in DR6. This way, it should be impossible to write detector which would say: "rootkit detected". The only thing the detector could do, is to say: "watch out for system reboot, if so assume rootkit in the system". Of course such detector is probably unacceptable on some production systems…

However, this is not an elegant solution. We need something more sophisticated.

[7] See next paragraph about polymorphism to complete the discussion of memory scanning for rootkit.

More intelligent solution would be to emulate Debug registers, when detected that some 3rd party software tries to use DR0 or DR1 (which are using by rootkit). In most cases this could probably be done by exploiting already hooked #PF handler. However this approach should be investigated a little bit more, especially several special cases should be considered: when 3rd party software would like to set DRx at:

- KiTrap01
- KiTrap0E
- NewDBHandler
- NewPFHandler

#DB handler algorithm

Here is the algorithm for simple #DB handler:

```
rootkit_DB() {
 if (DR6.BD==1) int3   // reboot :)

 if (restore0) {
     fake_bytes0 → [DR0]
     restore0 := false
     DR7 ← disable DR3 breakpoint
 }
 if (DR6.B0==1){ // (read) access to address held in DR0
     original_bytes0 → [DR0]
     restore0 := true
     DR3 := saved_EIP + offset_to_next_instruction(saved_EIP)
     DR7 ← enable DR3 breakpoint (on eXec)
     return;
 }

 if (restore1) {
     fake_bytes1 → [DR1]
     restore1 := false
     DR7 ← disable DR3 breakpoint
 }
 if (DR6.B1==1){ // access to addr held in DR1
     original_bytes1 → [DR1]
     restore1 := true
     DR3 := saved_EIP + offset_to_next_instruction(saved_EIP)
     DR7 ← enable DR3 breakpoint (on eXec)
     return;
 }
}
```

Hiding additional code and data: polymorphism

There is one more thing which we should take care of. Even though we probably be able to cheat about all the modification we made to the existing kernel code, we should be aware the we are adding some new code to the kernel memory. Of course, these new code, can look like dynamic buffer, but, when the administrator knows the exact version of the rootkit binary, she is able to detect the rootkit's additional code by just pattern searching… This is not good.

It should be noted that some of our new code cannot be held on marked pages (with *present* bit cleared). The example of this kind is our new PF handler.

It seems that the only remedy to this problem is to have polymorphic code generator to some rootkit's functions. Not all, however, only for those which cannot be held on the marked pages.

Polymorphic generator should be further investigated, some information about this subject can be found at [6].

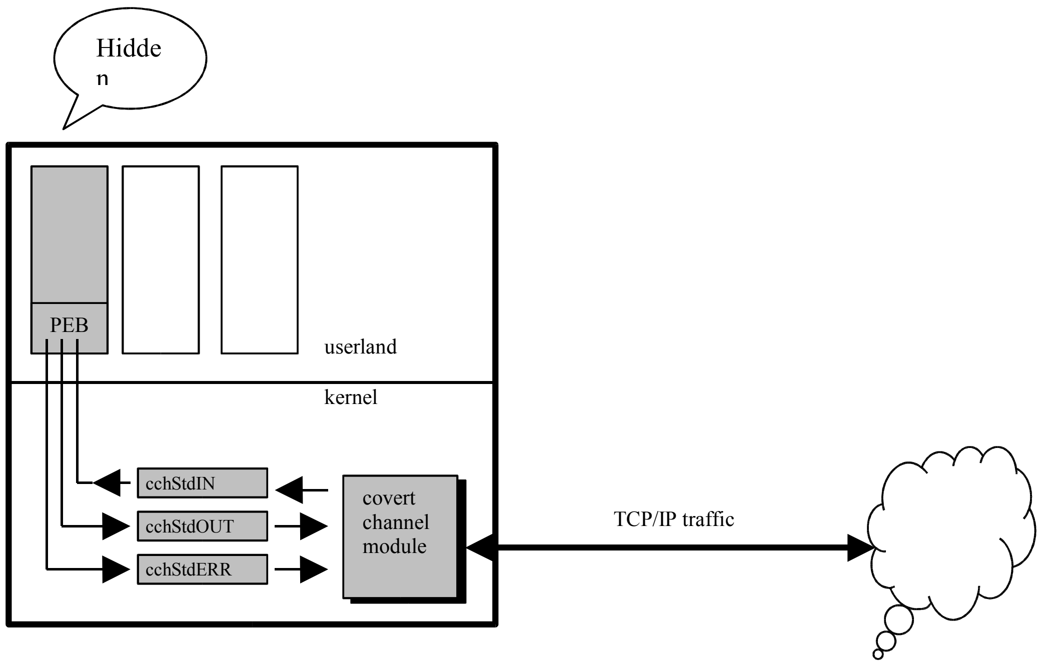
Covert Channel

Even best process hider would be useless without the possibility to communicate with the hidden processes. On the other hand, every communication effort can betray the hidden process. A good covert channel should be then implemented as a addition to the process hider.

There are currently some ideas around for implementation of TCP/IP based covert channel, like [7]. They should be further investigated. Network IDS resistance is the priority over the speed.

It would be desirable to define a good interface between covert channel module and the (hidden) process. Thanks to this, it would be possible to choose appropriate covert scheme depending on the role of the machine which is infected (WWW server, internal file server, workstation, etc…).

It is expected that the covert module will be actually some code residing in the kernel (something like NDIS driver). This code will take care of creating and receiving all network traffic which will be necessary to implement a given covert channel. This module should create three, unnamed, kernel objects: *cchStdIN*, *cchStdOUT* and *cchStdERR*. The PEB of the hidden process (actually *PEB.ProcessParameters.Std** fields) should be modified in such a way they will point to the objects created by covert module. This way no changes to the hidden application are needed (assuming that is a console application of course). The covert channel module can also be easy replaced by some other, depending on the covert channel requirements.



Putting it all together

The most convenient way to implement this rootkit will be a kernel driver. However it would be good to do it in such a way, that after unloading the driver rootkit will remain in the system. It means that we should use other method for allocating memory for additional code (like new #PF and #DB handlers). It should be investigated if it would be enough to just put the code in the allocated dynamic buffers.

It seems that userland client program for the rootkit is unnecessary. Such utility is never enough stealth. All the configuration activities should be done during compilation phase. This configuration includes such parameters like:

- process(es) to be hidden. This can be done by specifying the program name (like *st34lthcmd.exe*)
- the covert channel module to be used.
- covert channel module password.

Alternatively configuration can be done through the covert channel session (see [3] for similar concept). Additional kernel module will be needed, which will be seated between the Covert Channel Module and *ccmStd** object ends. It should be investigated if such feature is really necessary however.

Things such uploading or downloading files form the target machine, can be implemented by the hidden application, and are beyond the scope of the rootkit considerations.

The Future

This paper explains only some ideas behind implementation of the pretty stealth rootkit on Windows based systems. In most cases no proof of concept code has been written yet to check if the presented ideas are really implementable. More work should be done to investigate such big issues like covert channel design and good polymorphic generator for some rootkit functions.

References

[1] Holy_Father, *Hacker Defender rootkit*, http://www.rootkit.host.sk/.

[2] James R. Butler II, *Detecting Compromises of Core Subsystems and Kernel Functions in Windows NT/2000/XP*, M.S. thesis, University of Maryland, Baltimore County, 2002.

[3] Greg Hoglund, *NT ROOTKIT*, telnet://rootkit.com.

[4] James Butler, Jeffrey L. Undercoffer and John Pinkston, *HIDDEN PROCESSES: The Implication for Intrusion Detection*, Proceedings of the 2003 IEEE Workshop on Information Assurance, United States Military Academy, West Point, NY, June, 2003.

[5] Jan K. Rutkowski, *Advanced Windows 2000 Rootkit detection,* Black Hat USA, July 30-31, 2003.

[6] *29A* magazine, http://www.29a.host.sk/.

[7] Simple Nomad, *Covering Your Tracks*, Black Hat USA, July 30-31, 2003.