

## Defeating Honeybots: System Issues, Part 2

*Thorsten Holz, Frederic Raynal* 2005-04-06

### Introduction

This paper will explain how an attacker typically proceeds in order to attack a honeypot for fun and profit. In [part one](#) we compared honeypots to steganography and then looked at three common techniques for virtualizing honeypots. For each of these methods, which included User Mode Linux, VMware environments, and chroot/jail environments, we looked at weaknesses that lead to their detection. It was made clear that while each of these have their advantages, they can all be easily detected by an experienced hacker.

Now, in the second and final part of this series we discuss the detection of Sebek, the primary data capture tool for honeypot researchers. Then we'll look at other techniques available for detecting honeypots, such as x86-specific ones and time based analysis. Let's get moving.

### Detecting Sebek

Sebek [[ref 1](#), [ref 2](#)] is the primary data capture tool used by honeynet researchers to capture an attacker's activities on a honeypot. It is a client/server tool available for different OSes, including Linux, Windows and \*BSD.

All versions of Sebek work by hijacking the read() system call. Sebek is thus able to record all data accessed by a hacker in an unencrypted way, via read(). It can, for example, log SSH-sessions, recover files copied with SCP and record all passwords used by intruders. The recorded data is sent in a covert manner from the client to the Sebek server. This transmission is done by modifying the network stack of the kernel in order to hide outgoing packets so that an intruder cannot see them.

In a previous SecurityFocus article we showed that it is possible to detect the presence of Sebek using the network layer [[ref 3](#)]. Now we want to concentrate on system level issues with Sebek detection. We will first take a closer look at the Linux version 2.1.7 and then later show similar results for Sebek on other architectures.

### Detecting Sebek on Linux

The Linux version of Sebek is a loadable kernel module and works similar to a rootkit. It

overwrites the pointer to the `read()` system call in the system call table with a pointer to its own version. It is therefore able to access all data passed to the `read()` function and can then monitor this data.

Sebek hides its presence the same way that *adore*, a popular rootkit, does: it uses `cleaner.c`, a simple loadable kernel module that unlinks the last module in the kernel's module list. This unlinking prevents programs like `lsmod` to display the presence of Sebek on a host. But the "hidden" Sebek module still has a complete module header structure in memory. This structure can be found with very little effort and can be used to locate Sebek's code (such as its `cleanup_module()` function) and its variable storage. The tool `module_hunter.c` [ref 4] is able to locate hidden modules. It searches for patterns of module structures in the kernel's address space and is able to detect modules hidden via unlinking, as shown below:

```
# lsmod
Module                Size  Used by    Not tainted
ds                    7092    2
yenta_socket         10912    2
pcmcia_core           42912    0 [ds yenta_socket]
ipsec                 268100    0
usbcore              64972    1

# insmod module_hunter.o
# cat /proc/showmodules
# dmesg | tail -n 9
address                module

0xe0e6d000             usbcore size: 0xfdcc
0xe0e7e000             snd-pcmcia size: 0x50b4
0xe0e86000             ipsec size: 0x41744
0xe0ec9000             module_hunter size: 0x3ec
0xe0f2e000             pcmcia_core size: 0xa7a0
0xe0f3a000             yenta_socket size: 0x2aa0
0xe0f44000             ds size: 0x1bb4
```

If you compare the above output of `lsmod` with the output of `module_hunter`, you will notice two differences: The module with the name "module\_hunter" (in green) is new because we have

loaded this module. And the module "snd-pcmcia" (in red) is hidden and thus not displayed by `lsmod`, however `module_hunter` is able to find it.

Furthermore, it is possible to locate the address of Sebek's `cleanup_module()` function using this technique. With the knowledge of this address, disabling this piece of software on a honeypot is quite feasible. A simple kernel module that jumps to the memory location of `cleanup_module()`, and thus executes this function, is able to remove Sebek from the host. This works because Sebek reconstructs the pointer to the original `read()` system call (`ord` in the following code snippet) as shown below:

```
if(sct && ord){
    sct[__NR_read] = (unsigned long *)ord;
}
```

Now, after calling `cleanup_module()`, the system call table has its original layout and no further logging takes place.

A technique that is commonly used by rootkit detection tools can also be used to detect the presence of Sebek on a host. By looking at the system call table and analyzing the pointers to the various system calls, it is possible to detect a modified host. In an unmodified system call table, the pointers to the `read()` and `write()` system calls are adjacent. Because Sebek changes the pointer of the `read()` system call, this adjacency is no longer given. Thus analyzing the pointers of this two system calls can detect a modified system call table. An example of this is shown below:

**Unmodified system call table:**

```
sys_read   = 0xc0132ecc
sys_write  = 0xc0132fc8
```

**After loading Sebek:**

```
sys_read   = 0xc884e748
sys_write  = 0xc0132fc8
```

You can see that the pointer to the `read()` system calls points to a far distant memory location and can thus conclude that someone modified the system call table.

Sebek also modifies the network stack to hide its presence. It has to adjust some counters to conceal the transmission of the logging data. This modification can also be detected by a sophisticated attacker who compares the kernel's internal network counters and the output of `ifconfig` or other tools.

In addition to detecting Sebek, it is also possible to reconstruct some sensitive information from the application, such as the destination IP address it uses for logging and its so-called magic number. This is done by searching through the memory block allocated by Sebek and applying some heuristics. Briefly spoken, the heuristic searches for variables that look like configuration variables from Sebek. If, for example, `0x001c0de` is found in Sebek's memory area, the heuristic concludes that this could be the magic number. This cannot be the destination IP address, since `0x001c0de` translates to `240.1.192.222`, a reserved IP address. Similar heuristics are applied to all other variables found in memory. More information on this topic is also available. [ref 5]

Under the project name NoSEBrEaK [ref 6], a toolkit has been developed by the members of the German HoneyNet Project to show the limitations of current Sebek versions on Linux. Besides the techniques that were presented above, this project also includes a tool called Kebes. Kebes uses `mmap()` in a very clever way and avoids `read()` system calls. By `mmap()`ing everything directly into memory, this tool provides a method of working on a honeypot without calling `read()`. Kebes implements all basic functionality of a shell, including the reading and writing of files, secure deleting, direct executing of programs, and more. Furthermore, it implements an encrypted channel between the attacker and the honeypot. Therefore, all logging of network activity is useless since the operator of the honeynet does not have the appropriate key for decryption.

## Detecting Sebek on other architectures

Similar methods of detection are also applicable for Sebek on other architectures. First we will take a look at the Windows version. Sebek for Windows is a kernel driver that hooks all console and GUI `read()` operations to capture all data. By traversing `PsLoadedModuleList`, it is possible to detect if a kernel driver is "hidden". This is comparable to searching through memory on Linux looking for "hidden" modules. With the help of tools like `KProcCheck` [ref 7] it is possible to detect Sebek on Windows, as seen below:

```
C:\>kproccheck -d
KProcCheck Version 0.1 Proof-of-Concept by SIG^2 (www.security.org.sg)

80400000 - \WINNT\System32\ntoskrnl.exe
80062000 - \WINNT\System32\hal.dll
F7410000 - \WINNT\System32\BOOTVID.DLL
[...]
F7298000 - SEBEK.sys
[...]
```

Another way to detect Sebek is to look out for hooked APIs, a technique that is similar to looking for memory locations that have changed in the system call table on Linux. KProcCheck is also able to display the Native APIs that are hooked, as we can see:

```
C:\>kproccheck -t
KProcCheck Version 0.1 Proof-of-Concept by SIG^2 (www.security.org.sg)

Checks SDT for Hooked Native APIs

ZwClose                18 SEBEK.sys [F729A092]
ZwCreateFile           20 SEBEK.sys [F729A98C]
ZwCreateKey            23 SEBEK.sys [F729AD10]
ZwEnumerateKey         3C SEBEK.sys [F729AE02]
ZwEnumerateValueKey   3D SEBEK.sys [F729AA50]
ZwOpenFile             64 SEBEK.sys [F729A8E6]
ZwOpenKey              67 SEBEK.sys [F729AD88]
ZwQueryDirectoryFile  7D SEBEK.sys [F729A4CC]
ZwQuerySystemInformation 97 SEBEK.sys [F729A5F0]
ZwReadFile            A1 SEBEK.sys [F7299CF0]
ZwRequestWaitReplyPort B0 SEBEK.sys [F7299F14]
ZwSecureConnectPort   B8 SEBEK.sys [F7299FE6]
ZwWriteFile           ED SEBEK.sys [F7299D48]

Number of Service Table entries hooked = 13
```

## Disabling Sebek on Windows

Finally, disabling Sebek on Windows is also possible. Windows keeps a complete copy of the SDT ServiceTable within the kernel file `ntoskrnl.exe`, and thus it is possible to restore SDT ServiceTable by using `SDTRestore`. [ref 8] This is again similar to reconstructing the system call table in Linux.

Note that the work on detection and disabling on Sebek for Windows was done by Tan Chew Keong, who published two papers on this topic. [ref 9]

## Sebek on OpenBSD

As a last example, let's take a quick look at the OpenBSD version of Sebek. On OpenBSD, Sebek is a patch for the kernel and the functionality is similar to other Sebek versions. All operations that are passed through `read()` is recorded and sent in a covert manner to the logging host. But once again, the presence of Sebek can be detected by an attacker. By simply searching through the kernel file and disassembling "dofileread" it is possible to detect the presence of Sebek:

```
# echo "disassemble dofileread" | gdb -q /bsd | grep sebek
0xd01c9bdc <dofileread+292>:      call    0xd01c967c <sebeklog>
```

Furthermore, detecting Sebek via bpf fingerprinting and reconstructing sensible information about Sebek is also possible. You can read more about the system issues with Sebek on OpenBSD in a short study by the Droids Corporation. [ref 10]

## Other places to look for honeypots

Many people looking for honeypots have been interested in detecting the presence of one inside a virtual machine. While some detection tricks are specific to certain implementations, others depend on the nature of what we want to do with the high interaction honeypot.

## When a processor tells the truth

One way to detect the presence of a virtual machine is specific to an x86-based processor. In protected mode, all memory accesses pass through either the "global descriptor table" (GDT) or "local descriptor table" (LDT). The GDT and LDT contain segment descriptors that provide the base address, access rights, type, length, and usage information for each segment. The GDT is

used by all programs whereas the LDT can optionally be defined on a per-task basis to expand the addressable range of the specific task.

The "interrupt descriptor table" (IDT) is similar to the GDT and LDT, but it holds gate descriptors that provide access to interrupt and exception handlers. It is thus a direct replacement for the interrupt vector table used in x86-based systems.

The registers GDTR, LDTR and IDTR all contain the linear addresses and sizes of their respective tables. With the help of the three instructions SGDT, SIDT, and SLDT, the contents of these registers can be retrieved.

The SGDT instruction stores the contents of the GDTR in a 6-byte memory location. The SLDT instruction stores the segment selector from the LDTR in a 16 or 32-bit general-purpose register or memory location. The SIDT instruction stores the contents of the IDTR in a 6-byte memory location.

These instructions are normally only used by operating systems but are not privileged in the Intel architecture. This means that these instructions can be executed by a non-privileged process running in user-land (ring 3). Since an Intel processor only has one LDTR, IDTR, and GDTR, a problem arises when multiple operating systems try to use the same registers -- if two operating systems are running concurrently (one host OS and one OS inside a virtual machine), then the host OS must relocate the IDTR of the guest OS so that both do not conflict. But a process running inside a virtual machine can execute the SIDT instruction to retrieve the content of IDTR. Since this instruction does not require privileges and does not generate an exception, the host OS can not prevent this instruction. But the process running inside the virtual machine gets the relocated address of the IDT and can conclude that it is running inside a virtual machine.

More Information about the limitation of x86-based hardware for virtualization is also available. [[ref 11](#), [ref 12](#)].

## When time tells the truth

The best solutions for building a high interaction honeypot are to do with improving logging or using a virtual machine. These techniques do have a price, however. Each action performed by the intruder on the honeypot is *longer* than on a sane system, whereby

*longer* can have several meanings:

- More instructions are executed, either to log the true instruction, or to emulate it.
- More time is needed to execute the true instruction, because it is not the only instruction to be executed.

Hence, having reliable ways to measure either the number of instructions or the execution time also provides an efficient way to detect a hazardous honeypot environment.

A solution known as *execution path analysis* (EPA), based on a counter for executed instructions, has been given in Phrack by Jan K. Rutkowski. [ref 13] The principle of this method is to hook the syscall handler (`int 80`) and the debug exception handler (`int 1`) in the IDT (Interrupt Description Table). Then, by setting the TF bit (mask `0x100`) in the EFLAGS register, the new handlers are able to count each SIGTRAP generated when an instruction is executed. Initially proposed for Linux, it has been also ported to Windows. This port was not easy since Windows includes a way to protect the IDT. A recent article by Edgar Barbosa demonstrates how to circumvent EPA. [ref 14]

In the Phrack article, there is also a discussion about the x86 instruction `rdtsc`, which was an answer to Marcin Szymanek's problem. Jan K. Rutkowski claims that measuring syscall's time with `rdtsc` is not accurate enough due to the optimizations in new compilers. Nevertheless, G. Delalleau, another member of the French Honey net Project, chose to explore this further [ref 15] because EPA has several drawbacks:

- EPA requires some high level privileges (root and kernel space access)
- Modification in the system calls are not that covert: the DF bit can be detected as the change of the address for the `int 1`'s handler.

Hence, Delalleau proposed a solution to detect the changes in execution time based on specific registers and instructions provided by the CPUs (such as `rdtsc` on x86). The well-known problem with this, however, is that measurements can change greatly due to context switches, IRQ, and other things. Examining the mean time or the minimal execution time should be the first place to look. Please note that while we will keep using the general term *time* below, while we are in fact measuring ticks.

Delalleau gave a solution to make the time comparison more accurate, by doing many measurements so that they are statistically valid. During a learning phase, one evaluates the

distribution for execution times. Then, when one arrives on a suspect host, he can compare the reference distributions between the sane system and the suspect one. To compare distributions, Delalleau used the usual scalar product between the Fourier transform from both sets of measures (one needs to remove high frequencies to suppress the noise). The normalized scalar product of the sets will be close to 1, and will thus indicate a clean host (that is, if the reference set corresponds to a clean system of course), while it will be close to 0 otherwise.

This works quite well for detection, as root-kits hook functions and virtual machines each add several steps in the normal execution flow. Let's see two examples of this:

1. Detecting Sebek by measuring the execution time of the `read()` syscall is possible. On a system without Sebek, minimal time is around 8225 and scalar product 0.776282, whereas with Sebek, minimum time is 29999 and scalar product 0.009930. An explanation for this is that `read()` with Sebek is probably that long because each `read()` operation causes the sending of at least one UDP packet.
2. Detecting VMware is also very easy, and doing so causes an illegal instruction. This is because VMware's exceptions handler needs to check whether the exception must be handled by VMware itself or by a specific handler present in the process which caused the illegal instruction. The timing to handle an illegal instruction is 776mms on a sane system, while it increases to 2530mms on VMware.

A tool called `ktime` is available and can be obtained from [Gaël](#).

## Summary

The [first of this two-part series](#) compared honeypots to steganography and then looked at three common techniques for virtualizing honeypots. In this article we continued the discussion by looking at the Sebek data capture tool and how it can be detected. Then, we looked at other techniques for detecting honeypots, nearly exhausting the various detection methods that are available. As you have seen, a honeypot can be detected if a hacker is determined or experienced enough, and therefore one may question the value of a honeypot. Let's conclude the series by taking a step back to discuss these issues.

## Conclusion

There are two ways to build a high interaction honeypot, and both of these methods can be combined: using a virtual machine, or improving the logging capabilities of an existing system.

Currently, high interaction honeypots catch primarily novice hackers or "script kiddies." The *toolz* they use are not that clever, but they are extremely efficient. We can bet that they will soon embed honeypot fingerprinting technologies to ensure their own safeness and effectiveness. And it will be sufficient that only one person need design a tool such as this for the benefit of thousands of novice hackers.

In this article series we've seen that honeypots can be detected in many different environments. Does this mean that building high interaction honeypot is a useless proposition? A few years ago, port scans were the background noise of the pirates on the Internet, and were easily detected by firewalls. Some years later, these became vulnerabilities scanners which were detected by any IDS. Now, the noise of pirate chatter is recorded with honeypots -- automatic tools exploiting well-known flaws. This progression tells us it is already time to prepare the next generation of high interaction honeypots. Things are evolving quickly.

As Darwin says, that's life!

## References

[ref 1]

Sebek, by Edward Balas et al.

<http://www.honeynet.org/tools/sebek/>

[ref 2]

Know Your Enemy: Sebek

<http://www.honeynet.org/papers/sebek.pdf>

[ref3]

Defeating Honeybots: Network Issues, Parts 1 & 2

<http://www.securityfocus.com/infocus/1803>

<http://www.securityfocus.com/infocus/1805>

[ref 4]

Finding hidden kernel modules (the extrem way), by madsy

<http://www.phrack.org/show.php?p=61&a=3>

[ref 5]

Attacking Honeynets, by Maximillian Dornseif, Thorsten Holz, and Christian Klein

<http://www-i4.informatik.rwth-aachen.de/lufg/research/projects/honeynet/material/NoSEBrEaK-IAW.pdf>

[ref 6]

NoSEBrEaK Project, Kebes toolkit

<http://md.hudora.de/presentations/#nosebreak>

[ref 7]

KProcCheck

Win2K Kernel Hidden Process/Module Checker, by Tan Chew Keong

<http://www.security.org.sg/code/kproccheck.html>

[ref 8]

SDT Restore for Win2K/XP, by Tan Chew Keong

<http://www.security.org.sg/code/sdtrestore.html>

[ref 9]

Detecting Sebek Win32 Clients, by Tan Chew Keong

<http://www.security.org.sg/vuln/sebek215.html>

<http://www.security.org.sg/vuln/sebek215-2.html>

[ref 10]

Sebek2 client for OpenBSD, by Droids Corporation

<http://honeynet.droids-corp.org/download/sebek-opensbd.pdf>

[ref 11]

Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor

<http://www.cs.nps.navy.mil/people/faculty/irvine/publications/2000/VMM-usenix00-0611.pdf>

[ref 12]

Red Pill... or how to detect VMM using (almost) one CPU instruction, by Joanna Rutkowska

<http://invisiblethings.org/papers/redpill.html>

[ref 13]

Execution path analysis: finding kernel based rootkits, by J.K. Rutkowski

<http://www.phrack.org/show.php?p=59&mp;a=10>

[ref 14]

Avoiding Windows Rootkit Detection, Edgar Barbosa, 2004.

<http://www.rootkit.com/vault/Opc0de/bypassEPA.pdf>

[ref 15]

Mesure locale des temps d'exécution: application au contrôle d'intégrité et au fingerprinting, by G. Delalleau

SSTIC 2004 - [http://actes.sstic.org/SSTIC04/Fingerprinting\\_integrite\\_par\\_timing/](http://actes.sstic.org/SSTIC04/Fingerprinting_integrite_par_timing/)

## Credits

Thanks to Kelly Martin, Lance Spitzner, Dragos Ruiu, Maximillian Dornseif, Christian Klein, Felix Gärtner, Lutz Böhne, Laurent Oudot, Philippe Biondi, and folks from the German and the French Honey net Projects.

## About the authors

[Thorsten Holz](#) is a research student at the Laboratory for Dependable Distributed Systems at RWTH Aachen University. He will presumably graduate next fall and continue his studies as a Ph. D. student. His research interests include the practical aspects of secure systems, but he is also interested in more theoretical considerations of dependable systems. He is one of the founders of the German Honey net Project.

[Frédéric Raynal](#) is head of the Software Security Research and Development team at the Common Research Center (CRC) of EADS. He is also the Chief Editor of the first french magazine dealing with computer and information security (MISC), and Head of the Organisation Committee of [SSTIC](#) (Symposium sur la Sécurité des Technologies de l'Information et de la Communication). He worked on information hiding and cryptography as he earned his PhD. Now, he deals with (in)secure programming and security of operating systems. He also contributes to several open source projects and is part of the French Honey net Project.

[Privacy Statement](#)

Copyright 2006, SecurityFocus