T1B2 1555
Proceedings of the 2005 IEEE
Workshop on Information Assurance and Security
United States Military Academy, West Point, NY, 15–17 June 2005

Detecting Honeypots and other suspicious environments

Thorsten Holz Frederic Raynal

Abstract— **To learn more about attack patterns and attacker behavior, the concept of electronic decoys, i.e. network resources (computers, routers, switches, etc.) deployed to be probed, attacked, and compromised, is used in the area of IT security under the name *honeypots*. These electronic baits lure in attackers and help in assessment of vulnerabilities.**

Because honeypots are more and more deployed within computer networks, malicious attackers start to devise techniques to detect and circumvent these security tools. This paper will explain how an attacker typically proceeds in order to attack this kind of systems. We will introduce several techniques and present diverse tools and techniques which help attackers. In addition, we present several methods to detect suspicious environments (e.g. virtual machines and presence of debuggers). The article aims at showing the limitation of current honeypot-based research. After a brief theoretical introduction, we present several technical examples of different methodologies.

I. Introduction

Often we have a lack of precise information dealing with attacks on the Internet. In most cases, we just see the *results* of attacks against networks or specific computers. For example, after a successful attack we just *see* that the compromised computer attacks further computers within the network. But analyzing *how* the attacker proceeded is a difficult and time-consuming task. In addition, we do not have precise quantitative predications of attacks against computer systems and the tools, tactics, and motives involved in computer and network attacks are often not known in detail.

To change this, the concept of electronic decoys has been applied to the area of IT security recently. The term *honeypot* usually refers to an entity with certain features that make it especially attractive and can lure attackers into its vicinity. Honeypots are electronic bait, i.e. network resources (computers, routers, switches, etc.) deployed to be probed, attacked, and compromised. These systems run special software which permanently collects data about the system and greatly aids in post-incident computer and network forensics. A honeypot is usually a computer system with no conventional task in the network. This assumption aids in detection of incidents: Every interaction with the system is suspicious and could point to a possibly malicious action. This zero false-positives rates is a clear advantage of honeypots in contrast to intrusion detection systems (IDS). Several honeypots can be assembled into networks of honeypots called *honeynets*. Because of the wealth of data collected through them, honeynets are considered a useful tool to learn more about attack patterns and attacker behavior in communication networks. A detailed introduction to honeypots can for example be found in [1].

In contrast to this, malicious attackers (so called *blackhats*) try to devise new techniques to detect and circumvent honeypots and other suspicious environments. The attackers probably do not want that someone observes their action since this could lead to information leakage. Furthermore, they do not want to disclose their exploits and methods. For instance, if they intrude a system using a non-publicly known flaw (called a *0-day*), they do not want to share this knowledge since it will lose much of its value as soon as a patch is available. Moreover, once an attacker compromised a system, he wants to conceal his actions, whatever they can be: downloading and using new tools, chatting on IRC, and so on.

Very similar to this arms-race between people who run honeynets on the one side and blackhats on the other side is the area of steganography. It's goal is to hide the existence of a communication channel between several parties. Steganography came back to the front of the stage a few years ago, when Simmons introduces his prisoners problem [2]: Assume two prisoners are jailed in different cells. A warden has been authorized to bring messages from the one to the other. If messages are ciphered – which means the warden can not understand the content of the message – he will become suspicious, and the communication channel will be stopped. But if the prisoners have agreed on a code (for instance, a red sun on a painting means a thing, while a yellow sun means another thing), the message will not be noticed by the warden, and the prisoners have a chance to escape.

When deploying a honeypot, the goal is to capture lots of information about the activity of the attacker. Even if he notices that he is on a honeypot, learning how he noticed it is supposed to be a valuable information. This means that honeypots need to be covert, but not too covert.

Summarizing, steganography and honeypots share some characteristics: Mainly, once the existence of the honey-

holz@i4.informatik.rwth-aachen.de – Laboratory for Dependable Distributed Systems, RWTH Aachen University, Germany
frederic.raynal@eads.net – Laboratory for Security of Information Systems, EADS CRC, Paris, France f.raynal@miscmag.com – MISC Magazine, Paris, France

ISBN 0-7803-8572-1/$10.00

pot/communication channel is discovered by an attacker, the game is almost over. In both applications, the presence of something has to be hidden as good as possible. But there are always inevitably signs left. For example, the warden can examine the image and he will notice the differences between several pictures. For honeypots, the situation is similar: If an attacker watches out carefully for signs of deception, he will sooner or later find some.

In this paper we want to show how an attacker typically proceeds in order to attack or detect honeypots. We will introduce several techniques and present diverse tools and techniques which help attackers. In addition, we present several methods to detect suspicious environments (e.g. virtual machines and presence of debuggers). The article aims at showing the limitation of current honeypot-based research. After a brief theoretical introduction, we present several technical examples of different methodologies.

The paper is outlined as follows: Section II gives an overview of related work in the field of detection of honeypots. Several ways to detect honeypots and other suspicious environments are presented in section III. Directions of further work are outlined in section IV and we conclude this paper with section V.

II. Related work

Since honeypots are spreading all over networks, more and more people are interesting in defeating them. First issues where published in the year 2004 in fake releases of the well-known Phrack Magazine [3], [4]. In these articles, the author introduced several ways to fingerprint honeypots, either locally or remotely.

However, as *Sebek* [5], [6] is the primary data capture tool used by honeynet researchers to capture the attacker's activities on a honeypot, it focuses attention. In [7], the authors propose several ways to detect, disable and circumvent Sebek. In addition, they introduce a kind of shell called *Kebes* which is designed to avoid logging mechanisms installed by Sebek. While [7] focuses on Linux version of Sebek, [8] deals with the Windows version of Sebek. He uses some of his previous results to detect hidden process or to restore the Service Descriptor Table. Furthermore, [9] introduces several ways to detect the presence of Sebek on OpenBSD.

Some of the *high-interaction honeypots* (i.e. those where an attacker can connect to, and perform some actions – in contrast to *low-interaction honeypots* that just simulate a service) are based on virtual machines. Security of these virtual machines have been studied in [10], which demonstrates the limitations of the Pentium processor. Based on these results, [11] provides a short program to detect such an environment without needing any privileges.

III. Detecting Honeypots and other suspicious environments

A. User-mode Linux (UML)

Some have tried to used *User-mode Linux* (UML) as a honeypot [12], but first, let us recall what UML is. Basically, UML is a way to have a Linux kernel running in another Linux. We will call the initial Linux kernel the *host kernel* (or *host OS*), while the one started by the command `linux` will be called the *guest OS*. It runs "above" the host kernel, all in userland. Note that UML is only a hacked kernel, able to run in userland. Thus, you have to provide the filesystem containing your preferred Linux distribution.

By default, UML executes in *Tracing Thread* (TT) mode. One main thread `ptrace()`s each new process started in the guest OS. On the host OS, you can see this tracing with the help of ps:

```
host>> ps a
[...]
 1039 pts/6   S    0:00 linux [(tracing thread)]
 1044 pts/6   S    0:00 linux [(kernel thread)]
 1049 pts/6   S    0:00 linux [(kernel thread)]
 [...]
 1066 pts/6   S    0:00 linux [(kernel thread)]
 1068 pts/6   S    0:00 linux [/sbin/init]
 1268 pts/6   S    0:00 linux [ile]
 1272 pts/6   S    0:00 linux [/bin/sh]
 1348 pts/6   S    0:00 linux [dd]
[...]
```

You can identify the main thread (PID 1039) and several threads which are `ptrace()`d: Several kernel threads (PID 1044 – 1066), `init` (PID 1068), `ile` (PID 1268), a shell (PID 1272), and `dd` (PID 1348). You can retrieve a similar listing if `hostfs`, a module to mount a host OS directory into the UML filesystem, is available:

```
uml# mount -t hostfs /dev/hda1 /mnt
uml# find /mnt/proc -name exe | xargs ls -l
```

When used with default values, UML is not designed to be hidden as the output of `dmesg` shows:

```
uml>> dmesg
Linux version 2.6.10-rc2
...
Kernel command line: ubd0=[...]
...
Checking that ptrace can change system call
          numbers...OK
Checking syscall emulation patch for ptrace...
          missing
Checking that host ptys support output SIGIO...Yes
Checking that host ptys support SIGIO on close...
          No, enabling workaround
Checking for /dev/anon on the host...Not
```

```
         available (open failed with errno 2)
NET: Registered protocol family 16
mconsole (version 2) initialized on [...]mconsole
UML Audio Relay (host dsp = /dev/sound/dsp,
         host mixer = /dev/sound/mixer)
Netdevice 0 : TUN/TAP backend -
divert: allocating divert_blk for eth0
...
Initializing software serial port version 1
 /dev/ubd/disc0: unknown partition table
...
```

All lines in the above listing are specific to UML in default mode and thus allow fingerprinting. Another sign of UML is the usage of the TUN/TAP backend for the network device 0 (also included in the above listing). This is not that common on a real system and thus also allows the identification of UML.

One of the big issue with UML is that it does not use a real hard disk but a fake IDE device, called `/dev/ubd*`. Via looking at the file `/etc/fstab`, executing the command `mount`, or checking the directory `/dev/ubd/`, it is possible to notice the presence of an UML system. To hide that information, it is possible to start UML with the options `fake_ide` and `fakehd`. However, what is displayed may not be the truth as the major number identifying the devices `/dev/ubd*` is `98`(`0x62`), which is not the same as the one for IDE or SCSI drives.

UML can also be easily identified by taking a look at the `/proc` tree. Most of the entries in this directory show signs of UML as the following two examples show: In the first example, the file `/proc/cpuinfo`, which contains a collection of CPU and system architecture dependent items, gives us the information that this is a UML system in TT-mode. In the second examples, the content of `/proc/ksyms` tells us that this is a UML.

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : User Mode Linux
model name      : UML
mode            : tt
[...]

$ egrep "uml|honey" /proc/ksysms
a02eb408 uml_physmem
a02ed688 honeypot
```

In addition, the files `iomen`, `filesystems`, `interrupts`, and many others look suspicious and allow fingerprinting of UML. To counter this way of identifying UML, it is possible to use `hppfs` (Honeypot procfs, [13]) and customize the entries in the `/proc` hierarchy. However, this is a time-consuming and error-prone task.

Another place to look at is the address space of a process. The file `/proc/self/maps` contains the currently mapped memory regions and access permissions of the current process. On the host OS, the address space looks like:

```
host>> cat /proc/self/maps
08048000-0804c000 r-xp [...]  /bin/cat
0804c000-0804d000 rw-p [...]  /bin/cat
0804d000-0806e000 rw-p [...]
b7ca9000-b7ea9000 r--p [...]
               /usr/lib/locale/locale-archive
b7ea9000-b7eaa000 rw-p [...]
b7eaa000-b7fd3000 r-xp [...]
               /lib/tls/i686/cmov/libc-2.3.2.so
b7fd3000-b7fdb000 rw-p [...]
               /lib/tls/i686/cmov/libc-2.3.2.so
b7fdb000-b7fde000 rw-p [...]
b7fe9000-b7fea000 rw-p [...]
b7fea000-b8000000 r-xp [...]  /lib/ld-2.3.2.so
b8000000-b8001000 rw-p [...]  /lib/ld-2.3.2.so
bfffe000-c0000000 rw-p [...]
ffffe000-fffff000 ---p [...]
```

The first column shows the address space in the process that it occupies. The second column is a set of permissions (r = read, w = write, x = execute and p = private) and the third column in this listing is the pathname.

In contrast to that, the address space inside the guest OS looks like:

```
uml:~# cat /proc/self/maps
08048000-0804c000 r-xp [...]  /bin/cat
0804c000-0804d000 rw-p [...]  /bin/cat
0804d000-0806e000 rw-p [...]
40000000-40016000 r-xp [...]  /lib/ld-2.3.2.so
40016000-40017000 rw-p [...]  /lib/ld-2.3.2.so
40017000-40018000 rw-p [...]
4001b000-4014b000 r-xp [...]
                  /lib/tls/libc-2.3.2.so
4014b000-40154000 rw-p [...]
                  /lib/tls/libc-2.3.2.so
40154000-40156000 rw-p [...]
9ffff000-a0000000 rw-p [...]
beffe000-befff000 ---p [...]
```

What is not that common is the top-most address, which indicates the end of the stack. The mapping of the dynamic libraries is not relevant for this example. Depending on the amount of memory available on the host, the end of the stack is usually `0xc0000000`. However, in the guest OS it is `0xbefff000`. In fact, the address space between `0xbefff000` and `0xc0000000` inside the UML system contains the mapping of the UML kernel. This means that each process can access, change, or do whatever it wants with the UML kernel.

Summarizing, it is pretty easy to fingerprint the presence of UML. We have implemented the techniques outlined above in a little tool called `UMLfp`.

To fix most of these problems, it is possible to start UML either with the argument `honeypot` or with the `skas` mode (Separate Kernel Address Space, [14]). However, having `skas` mode running is not that easy, and the host kernel is really not stable. During tests, we had especially problems with pending processes which lead to reboots in our setup.

B. VMware

VMware [15] is a very efficient virtual machine software which provides a virtual x86 hardware. Thus, it is possible to install (almost) any operating system on VMware, for example Linux, Windows or Solaris 10. These operating systems are isolated in secure virtual machines and the VMware virtualization layer maps the physical hardware resources to the virtual machine's resources, so each virtual machine has its own CPU, memory, disks, I/O devices, and others.

So, the first step to detect VMware is to look at the hardware since it is supposed to emulate it. Prior to version 4.5, there are some specific pieces of hardware that are not configurable:
1. the video card: `VMware Inc [VMware SVGA II] PCI Display Adapter`
2. the network card: `Advanced Micro Devices [AMD] 79c970 [PCnet 32 LANCE] (rev 10)`
3. the name of IDE and SCSI devices: `VMware Virtual IDE Hard Drive`, `NECVMWar VMware IDE CDR10`, `VMware SCSI Controller`

It is possible to patch the VMware binary to change these default values. Kostya Kortchinsky from the French Honeynet Project has written a patch which is able to set these values to some other values [16].

It is also possible to identify a running VMware in default mode by looking at the MAC address of the network interface [17]. The following ranges of MAC addresses are assigned to VMWare, Inc by IEEE [18]:

```
00-05-69-xx-xx-xx
00-0C-29-xx-xx-xx
00-50-56-xx-xx-xx
```

The MAC address of the network interface can be retrieved by looking at the cached MAC addresses with the command `arp -a` or by looking at the data related to the interface (Unix systems: `ifconfig`, Windows systems: `ipconfig /all`). Thus it is possible to fingerprint VMware this way.

Furthermore. the VMware binary has an I/O backdoor. This backdoor is used to configure VMware during runtime. An analysis of *Agobot*, an IRC-controlled backdoor with network spreading capabilities, revealed that this I/O backdoor of VMware is used for detection. The following sequence is used to call backdoor functions:

```
mov eax, VMWARE_MAGIC ; 0x564D5868
mov ebx, b            ; <parameter of command>
```

Command number	Description
05h	Set current mouse cursor position
07h	Read data from host's clipboard
09h	Send data to host's clipboard
0Ah	Get VMware version
0Bh	Get device information

Fig. 1. Possible commands to execute via VMware backdoor

```
mov ecx, c            ; <number of command>
mov edx, VMWARE_PORT  ; 0x5658

in eax, dx
```

At first, register `EAX` is loaded with a magic number that is used to "authenticate" the backdoor commands. Register `EBX` stores parameters for the commands and in register `ECX` the command itself is loaded. Table 1 gives an overview over some possible commands [19]. In total, there are at least 15 implemented commands.

Register `DX` stores the I/O backdoor port and with the help of the `IN` instruction, the backdoor command gets executed finally. So with the help of the VMware I/O backdoor, it is possible to interfere with a running VMware.

The patch by Kostya Kortchinsky [16] can change the magic number and thus somewhat "hide" the backdoor from an attacker.

C. Detecting additional lines of defense: chroot and jails

`chroot()` has never been designed for security, but is considered as a necessity as soon as one wants to protect a sensitive server. Detecting of a `chroot` environment – or even circumventing it – is not really difficult. Unless the `chroot` directory is on a specific partition, and placed on top of it, the inode numbers are not those expected at a real root directory. This information can be retrieved with the `ls -i` command:

```
# ls -ialgG /
2 drwxr-xr-x  24  4096 2004-11-30 08:14 .
2 drwxr-xr-x  24  4096 2004-11-30 08:14 ..
[...]
```

Here, the directories inodes of `.` and `..` are the same, and are equal to 2, which is the normal value for a root directory on a partition. In the current directory, the command returns the following information:

```
# ls -ialgG .
1553552 drwxr-xr-x  6  4096 2004-12-14 13:58 .
6657574 drwxr-xr-x  6  4096 2004-12-12 16:25 ..
```

When using `chroot` with a shell in the current directory, `ls -i` returns the same inodes numbers for `.` and `..`:

```
# chroot . /bin/busybox
BusyBox v0.60.5 (2004.10.29-22:08+0000)
```

multi-call binary

```
# ls -ialgG
1553552 drwxr-xr-x   6  4096 Dec 14 12:58 .
1553552 drwxr-xr-x   6  4096 Dec 14 12:58 ..
```

While the `..` has been changed to match the `.` directory, it is still not the expected value of 2.

But there is much more to do in a `chroot` environment. For instance, it is possible to send signals to any process outside the `chroot()`, or even attach to outside processes with `ptrace()`. Since `ptrace()` can be executed from inside the `chroot` environment on any process outside the `chroot()`, it can be used by an attacker to inject whatever he wants on the host. Such evasions are also possible through `mount()`, `fchdir()`, `sysctl()` and many other commands [20].

So, when thinking about virtual environments and security, `chroot()` is definitely not to use. Therefore, FreeBSD enforces confinement based on `chroot` to provide another mechanism, designed to be more reliable: the `jail()`. A `jail` creates a virtual host, bound to an IP address, with its own tools, users, and so on. This is very convenient for virtual hosting, and could be for honeypots, too. However, since it is more reliable, it is not very covert. There are several tests to fingerprint a `jail`:
1. All processes in a `jail` have a specific "J" flag:

```
jail# ps
   PID  TT  STAT      TIME COMMAND
  6908  p0  SJ     0:00.02 /bin/sh
  6910  p0  R+J    0:00.00 ps
```

In addition, the PIDs do not increase in the usual way inside a `jail`.
2. The inode number of the root directory is not 2 as expected on a real system.
3. By default, raw sockets are forbidden:

```
jail# ping -c 3 miscmag.com
ping: socket: Operation not permitted
```

Note that this is configurable in the latest FreeBSD release.
4. Sniffing in a `jail` environments gives access to all traffic that comes through the device. This is a design issue since a `jail` is usually built as an alias on a real device.

In this section, we focused on detection if we are jailed in a confined environment. However, is this a real issue? Learning we are on a "restricted host" is not that important as such systems are spreading all around Internet. However, the real issue deals with the leaking from the guest system to the host system. And currently, there are very few (if any) systems that have proved to be well confined.

D. Timing issues

The main solutions to build a high interaction honeypot are improving logging or using a virtual machine. These techniques do have a price: each action performed by the intruder on the honeypot is *longer* than on a sane system. Longer can have several meanings:
1. More instructions are executed. Either to log the true instruction, or to emulate it.
2. 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 environment.

A solution, Execution path analysis (EPA), based on a counter for executed instructions, has been given in phrack 59 by Jan K. Rutkowski [21]. The principle is to hook the syscall handler (int 80) and debug exception handler (int 1) in the IDT (Interrupt Description Table). Then, by setting the TF bit (mask 0x100) in EFLAGS register, the new handlers are able to count each SIGTRAP generated when an instruction is executed. Initially proposed for Linux, it has been ported to Windows, too. This was not easy since Windows includes a way to protect the IDT. A recent article by Edgar Barbosa [22] demonstrates how to circumvent EPA. In phrack 59 [21], there is also a discussion about the x86 instruction `rdtsc` which is an answer to Marcin Szymanek. Jan K. Rutkowski claims that measuring syscall's time with `rdtsc` is not accurate enough due to optimizations in new compilers.

Nevertheless, G. Delalleau, another member of the French Honeynet Project, chooses to explore this further [23] because EPA has several drawbacks:
1. It requires some high privileges (root and kernel space access)
2. 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, he proposed a solution to detect the changes in execution time based on specific registers and instructions provided by the CPUs (e.g. `rdtsc` on x86-based hardware). The well-known problem is that measurements can change greatly due to context switches, IRQ, and other things. Looking at mean time or minimal execution time is the first place to look at. Please note that we will keep using the term *time* below, while we are in fact measuring ticks.

Delalleau proposed a solution to make the time comparison more accurate [23]: 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, the reference distributions between the sane system and the suspected one must be compared. Delalleau proposed to use the usual scalar product between the Fourier transform of the both sets of measures for comparison. It is necessary to remove high frequencies to suppress the noise. The normalized scalar product of the sets will be close to 1, and will thus indicates a clean host (if the reference set corresponds to a clean system of course), while it will be close to 0 on the contrary.

This works quite well because rootkits hook functions and virtual machines add several steps in the normal execution flow:
1. Detecting Sebek by measuring execution time of the `read()` system call. On a system without Sebek, minimal time is around 8225 and the scalar product is 0.776282. In contrast to this, a system with Sebek has minimal time of 29999 and scalar product of 0.009930. An explanation is probably that `read()` with Sebek is that long because each `read()` operation causes the sending of at least one UDP packet.
2. Detecting VMware is also very easy causing an illegal instruction as the VMware's exceptions handler needs to check whether the exception must be handled by VMware itself or by a specific handler present if 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 inside a running VMware.

E. Detecting Debuggers

Debugging is a feature provided by the processor (CPU) and managed by the operating system (OS). Most recent CPU and OS give several ways to supervise the way a process is running. Some features are available from ring 3 (*user mode* or *user space*), others are restricted to ring 0 (*supervisor mode* or *kernel space*). In this section, we will mainly focus on x86 architecture, and on Linux and Windows as OS.

Note that we will only deal with debugging here, but it should also be combined with reverse engineering techniques so that the analysis of the binary itself do not give any information. Obfuscation techniques include ciphering, dis-aligning the instructions, headers modification, junk code, and many others which will not be detailed.

Debugging is a very efficient way to learn about a process activity (even if this is not the only solution). As soon as a developer wants to protect his software, he can include in the instructions flow some mechanisms to prevent debugging. This is possible because debugging is a very low level feature, which makes it quite easy to detect. Firstly, we will introduce generic ways to debug a process, and then focus on specific techniques and tools.

The general way to trace process under Unix is to use the system call `ptrace()`. It allows a process to attach another and access all of its memory: data, instructions, and other information. There exists a very easy way for a process to check whether it is `ptrace()`d or not:

```
#include <sys/ptrace.h>
#include <stdio.h>

main()
{
    long int err;
    err = ptrace(PTRACE_TRACEME, 0, NULL, NULL);
    if (!err) printf("not traced\n");
    else perror("ptrace()");
}
```

Calling `ptrace(PTRACE_TRACEME, 0, NULL, NULL)` will force a process to attempt to `ptrace()` itself. Since a process can only be `ptrace()`d once, this will fail if a process is currently debugged by another process.

Under Windows, there is an API called `IsDebuggerPresent()`: with Windows NT, it searches in the Process Environment Block (PEB) for the field `IsDebugged`. It works differently with Windows 9x since there is no PEB. So, a program can use this API to check the presence of a ring 3 debugger. In fact, this API is not called directly, but the corresponding assembler code is embedded in the program.

These `ptrace()` and `IsDebuggerPresent()` functions are quite high level APIs provided by the OS. However, they are built on features of the processor / main board.

Furthermore, there are *software breakpoints*. They are caused by the `int 3` assembler instruction, whose specific opcode is `0xCC`. It's main default regarding our topic is that it is a destructive way to debug a program: the user has to replace an opcode in the memory section containing the instructions (usually referred as the code or text section). Hence, a program which contains and checks constantly its own checksum or cryptographic hash will detect the modification, and can stop or do whatever the programmer wants. For example, he can set a specific handler for this interruption in the Interrupt Descriptor Table (IDT) if the program is running under Windows 9x. This is no more possible with the latest releases of Windows nor with Linux as writing to the IDT requires to be in ring 0. A less computationally expensive way to prevent software breakpoints is to scan the memory for opcodes `0xCC`.

A further analysis of Agobot, an IRC-controlled backdoor with network spreading capabilities, showed that it does not only include functions to detect the presence of VMware, but also detect the presence of debuggers and breakpoints. The following code is used to detect software breakpoints:

```
mov esi, address ; load function address
mov al, [esi]    ; load the opcode
cmp al, 0xCC     ; check if the opcode is 0xCC
je BPXed         ; yes, there is a breakpoint
                 ; jump to return true
xor eax, eax     ; false,
jmp NOBPX        ; no breakpoint
BPXed:
  mov eax, 1     ; breakpoint found
NOBPX:
```

Another way to debug a program is to trace it step by step. This is done by controlling the 8th bit of the `EFLAGS` registers, which is called `TRAP`. When it is set to 1, the

processor initiates the `int 1`. Thus, a process can check easily it's `TRAP` bit by accessing the `EFLAGS` register though the `pushf` instruction.

All x86-based processors have also seven specific registers designed for debugging: `DR7` is a control register, `DR6` a status register, `DR5` and `DR4` are reserved and `DR3`, `DR2`, `DR1`, and `DR0` can contain an address to be supervised. The user just has to set the address which he wants to supervise in one of the address register, and chose what kind of operation (read or write) he wants to supervise using the control register `DR7`. As this is a privileged operation, it must be performed by the kernel itself (ring level 0) using a `movl` instruction. Hence, under Linux, you need to use the system call `ptrace()` to access these registers through the commands `PTRACE_PEEKUSR` and `PTRACE_POKEUSR`. Calling this will cause a system call which will bring the arguments in the ring 0 before accessing the registers. Thus, to prevent the use of debug registers, a program just needs to call `ptrace()` to set them to 0. Under Windows, it is feasible to change these registers in an exception handler. The programmer set a specific handler in the *Structured Exception Handling* (SEH) and cause an error in the code (e.g. a division by 0). When context-switching to the handler, debug registers are saved on the user stack. Hence, it is possible to read and write these saved values, which will be restored to the registers by the kernel when the handler will be over.

Under Windows, some softwares also embed a detection step for some common debuggers, like OllyDbg in ring 3 or SoftIce in ring 0. There are many solutions to detect their presence on the system. As a short example, an excerpt from the source code of Agobot shows a possible way to detect the presence of OllyDbg:

```
push 0x00
push caption          ; char *caption="DAEMON"

mov eax, fs:[30h]     ; pointer to PEB
movzx eax, byte ptr[eax+0x2]
or al,al
jz normal_
jmp out_
normal_:              ; return false,
  xor eax, eax        ; no debugger
  leave
  ret
out_:                 ; return true,
  mov eax, 0x1        ; debugger detected
  leave
  ret
```

Usually, when a debugger is running, the only protection for the attacker is to detect it's presence before performing anything and then escaping. However it can be much more fun if such a debugger contains a flaw, and thus be exploited by the reversed program. That way, it can have its code executed on the system without being supervised [24].

IV. Further work

As our research has shown, there are several ways to fingerprint current honeypot-related technologies. Furthermore it is possible to detect other suspicious environments and we showed that current malware already implements techniques to do so. So in the future, we need to develop existing tools further and improve their stealthiness, e.g. by removing additional signs of the emulator itself in the case of UML or VMware.

Another area of further research aims at developing new kinds of honeypots. Any fisherman knows it: to catch a specific fish, one needs a specific bait. Currently, the deployed honeypots are designed to catch generic attacks, for example worms and viruses, script kiddies using automatic tools, and so on. To catch advanced threats, we will probably need new types of honeypots.

A possible idea for such new types of honeypots include *client-side honeypots*: Since we see more and more attackers exploiting holes in client programs (e.g. via exploits in Microsoft's Internet Explorer), the honeypots have to further evolve. As clients depend on the server they are working with, we need to design client-side honeypots according to the protocol and what we want to catch.

We differentiate between two of client-side honeypots. On the one hand, these type of honeypots can be *active*. This is the usual behavior, since they connect to a given server, send some commands, and get back the results. In fact, active clients are *synchronous* (e.g. web browsers). On the other hand, some are *passive*, waiting for an event to happen. Those are *asynchronous* (e.g. mail clients), which means we have to find a way to trigger that event.

For synchronous client-side honeypots, a possible way of further research would be the development of a *web-based honeyclient*. This honeypot would aim at finding servers compromising the browser. The first step of this methodoloy will be to find sites attacking web browsers, and then understand what kind of attack it is. Finding the sites may not be that difficult using the same tricks as some worms do right now: `google.com`. May be classifying the results obtained by keyword may be interesting (`warez`, `sex`, `casino`, and so on).

The web-based honeyclient can be the target of different kinds of attacks:

- To install an IRC bot: the goal is to install an Internet Relay Chat (IRC) bot, so that it becomes part of a botnet and can be remotely controlled.
- To install a "proxy": the goal is to take control of the host and install a `SOCKS` proxy or an IRC bouncer.
- To install a spyware: the goal is to install spyware which will capture sensitive information, and install additional and malicious software on the victim's computer.

• To retrieve sensitive information from the victim's machine, for example credit cards numbers, passwords, or cookies (*identity theft*).

The web-based honeyclient has to perform an integrity check of the whole system after interacting with a specific website to determine if it has been compromised. Via montoring of file-system activity, monitoring of registry-modifications, and a couple of other operations, this can be achieved. To be valuable, the tests can send different `user-agent` strings. This could make the administrator of the site suspicious if he analyzes his log-files regularly. Thus, to prevent such a detection, it could probably be very useful to use some anonymizing devices.

Note that all these tests should be performed for MS Windows, with multiple browsers as it is currently the privileged target. However, if the tool is well written, tests must also been performed on other OS.

For asynchronous client-side honeypots, also several approaches for further research can be considered:

• IRC-based honeyclients that join a specific IRC server and channel (e.g. #warez, #1337). Then they just idle in this channel or throw in random quotes.

• Instant messenger-based honeyclients (e.g. AIM, ICQ, MSN, ...) that connect to the network and interpret received messages

• Mail-based honeyclients that download e-mails, analyze the content and click on links (thus being very similar to web-based honeyclients).

• Peer-to-Peer (p2p) based honeyclients that randomly download files from p2p-networks and execute it.

Again, these types of honeypots have to regularly check their own consistency and detect changes. This way, they can notice if they were exploited by malicious servers or other attackers.

V. Conclusions

There are two ways to build a high interaction honeypot, which can be combined: using a virtual machine, or improving the logging capabilities of a system. Currently, high interaction honeypots mainly catch script kiddies. The tools they use are not that clever, but are extremely efficient. We can bet that they will soon embed fingerprinting technologies to ensure their own safeness. With the fingerprinting techniques included in Agobot, this has already begun. And it will be sufficient if only one person decides to write functions for fingerprinting honeypots and other suspicious environments – thousands of kiddies benefit from these techniques and add them to their toolkit.

Does that mean building high interaction honeypot is useless? A few years ago, port scans were the background noise of the attackers in the Internet, and detected by firewalls. Some years later, it were vulnerability scanners, which were detected by IDS. Now, the noise is recorded with these honeypots: automatic tools exploiting well known flaws. This tells us it is already time to prepare the next generation of high interaction honeypots. Things are evolving quickly. Presumably the existing honeypots can be developed further to observe advanced threats, so the arms-race continues.

We want to thank all people from the French and German Honeynet Project who helped in our research. Special thanks go to Laurent Oudot and Gael Delalleau.

Thorsten Holz was supported by the Deutsche Forschungsgemeinschaft (DFG) as a research student in the DFG-Graduiertenkolleg "Software für mobile Kommunikationssysteme" at RWTH Aachen University.

References

[1] M. Dornseif, F. C. Gärtner, and T. Holz, "Vulnerability assessment using honeypots," *Praxis der Informationsverarbeitung und Kommunikation (PIK)*, vol. 4, no. 27, pp. 195–201, 2004.

[2] G. J. Simmons, "The prisoners' problem and the subliminal channel," in *Advances in Cryptology*, pp. 51–67, 1984.

[3] J. Corey, "Local honeypot identification." http://www.phrack.org/unoffical/p62/p62-0x07.txt.

[4] J. Corey, "Advanced honey pot identification." http://www.phrack.org/unoffical/p63/p63-0x09.txt.

[5] "Sebek." Internet: `http://honeynet.org/papers/honeynet/tools/sebek/`, 2004.

[6] The Honeynet Project, "Know your Enemy: Sebek," November 2003. `http://www.honeynet.org/papers/sebek.pdf`.

[7] M. Dornseif, T. Holz, and C. Klein, "Nosebreak - attacking honeynets," in *Proceedings of 5th Annual IEEE Information Assurance Workshop*, 2004.

[8] T. C. Keong, "Detecting sebek win32 client." http://www.security.org.sg/vuln/sebek215.html and http://www.security.org.sg/vuln/sebek215-2.html.

[9] D. Corporation, "Sebek2 client for openbsd." http://honeynet.droids-corp.org/download/sebek-openbsd.pdf.

[10] J. S. Robin and C. E. Irvine, "Analysis of the intel pentium's ability to support a secure virtual machine monitor," in *Proceedings of 9th USENIX Security Symposium*, 2000.

[11] J. Rutkowska, "Red pill... or how to detect vmm using (almost) one cpu." http://invisiblethings.org/papers/redpill.html.

[12] "Know your enemy: Learning with user-mode linux." http://www.honeynet.org/papers/uml/.

[13] "Honeypot procfs." http://user-mode-linux.sourceforge.net/hppfs.html.

[14] "Separate kernel address space & uml." http://user-mode-linux.sourceforge.net/skas.html.

[15] "Vmware homepage." Internet: `http://www.vmware.com/`.

[16] K. Kortchinsky, "Patch for vmware." http://honeynet.rstack.org/tools/vmpatch.c.

[17] T. Holz and L. Oudot, "Defeating honeypots: Network issues." http://www.securityfocus.com/infocus/1803 and http://www.securityfocus.com/infocus/1805.

[18] "Ieee standards." Internet: `http://standards.ieee.org/regauth/oui/oui.txt`.

[19] "Vmware backdoor i/o port." Internet: `http://chitchat.at.infoseek.co.jp/vmware/backdoor.html`.

[20] B. Spengler, "chroot(), sécurité illusoire ou illusion de sécurité ?." http://www.miscmag.com/articles/index.php3?page=1008 MISC 9 - Sept./Oct. 2003.

[21] J. K. Rutkowski, "Execution path analysis: finding kernel based rootkits." http://www.phrack.org/show.php?p=59&a=10.

[22] E. Barbosa, "Avoiding windows rootkit detection," 2004. http://www.rootkit.com/vault/Opc0de/bypassEPA.pdf.

[23] G. Delalleau, "Mesure locale des temps d'exécution: application au contrôle d'intégrité et au fingerprinting." SSTIC 2004: http://actes.sstic.org/SSTIC04/Fingerprinting_integrite_par_timing/.

[24] N. Brulez, "Scan of the month 33: Anti reverse engineering uncovered." http://www.honeynet.org/scans/scan33/nico/.