Debug register rootkits
A study of malicious use of the IA-32 debug registers

May 2012

Authors: Emil Persson, Joel Mattsson
Supervisor: Ewa Osekowska

School of Computing
Blekinge Institute of Technology
This thesis is submitted to the School of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science. The thesis is equivalent to 20 weeks of half time studies.

Contact Information:

Authors:
Emil Persson
E-mail: empc08@student.bth.se

Joel Mattsson
E-mail: jomf09@student.bth.se

University advisor:
Ewa Osekowska
E-mail: ewos10@student.bth.se

School of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona
Sweden

Internet: www.bth.se/com
Phone: +46 455 38 50 00
Fax: +46 455 38 50 57
Abstract

The debug register rootkit is a special type of rootkit that has existed for over a decade, and is told to be undetectable by any scanning tools. It exploits the debug registers in Intel’s IA-32 processor architecture. This paper investigates the debug register rootkit to find out why it is considered a threat, and which malware removal tools have implemented detection algorithms against this threat. By implementing and running a debug register rootkit against the most popular Linux tools, new conclusions about the protection of the Linux system can be reached.

Recently, debug register rootkits were found on Windows as well. This project intends to bring knowledge about the problem and investigate if there are any threats.

Our study has shown that still after 12 years, the most popular tools for the Linux operating system have not implemented any detection algorithms against this threat. The security industry may need to prepare for this threat in case it is spread further.

**Keywords:** Debug register, rootkit, IA-32, memory forging, Linux
Acknowledgements

We would like to thank our supervisor Ewa Osekowska for all her assistance regarding structure, design and grammar corrections in this thesis. She has shown great devotion in helping us complete our goals.
# Contents

1. **Introduction** ........................................................................................................... 1  
   1.1 Related work ........................................................................................................... 2  
   1.2 Aims and objectives ............................................................................................... 3  
   1.3 Research questions ............................................................................................... 3  
   1.4 Hypothesis .............................................................................................................. 4  
   1.5 Methodology .......................................................................................................... 4  
   1.6 Disposition ............................................................................................................. 4  
   1.7 Conventions ............................................................................................................ 5  

2. **Background** ............................................................................................................. 5  
   2.1 The Linux Kernel .................................................................................................... 5  
      2.1.1 Introduction ....................................................................................................... 5  
      2.1.2 The Intel X86 protection rings .......................................................................... 6  
      2.1.3 Interrupts and exceptions ................................................................................. 6  
      2.1.4 The Interrupt Descriptor Table ......................................................................... 7  
      2.1.5 The system call table ....................................................................................... 8  
      2.1.6 Kernel modules ............................................................................................... 9  
      2.1.7 Summary ....................................................................................................... 10  
   2.2 The debug registers ............................................................................................... 11  
      2.2.1 Introduction .................................................................................................... 11  
      2.2.2 DR0-DR3 ....................................................................................................... 11  
      2.2.3 DR4-DR5 ....................................................................................................... 12  
      2.2.4 DR6 .............................................................................................................. 12  
      2.2.5 DR7 .............................................................................................................. 13  
      2.2.6 Summary .................................................................................................... 14  
   2.3 Hooking ................................................................................................................ 14  
      2.3.1 Introduction .................................................................................................... 14  
      2.3.2 Hooks and rootkits ....................................................................................... 15  
   2.4 The Open Source Linux 2.6 Rootkit .................................................................... 15  
      2.4.1 Introduction .................................................................................................... 15  
      2.4.2 The debug register rootkit ............................................................................ 16  

3. **Experiment** ............................................................................................................. 19  
   3.1 Introduction .......................................................................................................... 19  
   3.2 Tools ....................................................................................................................... 19
3.3 The experiment..................................................................................................................20
3.3.1 Rootkit Hunter ..............................................................................................................21
3.3.2 Chkrootkit ....................................................................................................................22
3.3.3 AIDE .............................................................................................................................22
3.3.4 ClamAV ........................................................................................................................24
3.3.5 McAfee VirusScan Enterprise .....................................................................................24
4 Results .......................................................................................................................................25
4.1 Rootkit Hunter ....................................................................................................................25
4.2 chkrootkit ..........................................................................................................................26
4.3 AIDE .................................................................................................................................26
4.4 ClamAV .............................................................................................................................27
4.5 McAfee VirusScan Enterprise .........................................................................................27
5 Discussion ...............................................................................................................................27
6 Conclusions .............................................................................................................................28
7 Future Work .............................................................................................................................29
References ......................................................................................................................................30
1 Introduction

The computer security engineering field deals with prevention, detection, and response to various computer threats. Some common problems in the scientific area include improving authentication, networking security and malware (malicious software) prevention/detection. This report will lay emphasis on malware.

The use of malware is increasing every day. Depending on the purpose of the attacks (which can vary from spreading advertisements to compromising computer systems) different types of malware are used to carry them out. Viruses, worms, Trojan horses, and botnets may be some of the terms that are commonly used to distinguish between different malware. There is, however, a special kind of malicious software that is often treated differently from the rest: the rootkit.

A rootkit has only one purpose, and it is to keep the attacker’s presence hidden in an already compromised system. It can be defined as follows: “RootKits are Trojan horse backdoor tools that modify existing operating system software so that an attacker can keep access to and hide on a machine” [1].

Take notice of the word “keep” in the citation above. It is important to mention that the rootkit does not deal with intrusion into a system of any kind. It is always assumed that the system already has been compromised and that administrator or root user privileges have been attained.

A rootkit has many ways of using the infected system maliciously, e.g. by hiding processes, files, directories, sockets, executing programs, creating child processes, killing processes, and many more. Used in the right way it may give an attacker full control of the compromised system. This is often made remotely through a shell, therefore allowing an attacker to control a system from their own computer.

Over the years of development, two different types of rootkits have emerged: the user-mode rootkit and the kernel-mode rootkit. The rootkits do the same thing, regardless of the technique used. It is the means that differ. The difference lies in which files they manipulate. If they are modifying executable files in the operating system that also a user could execute, they are called a user-mode rootkits, and if they modify an operating system file within the kernel they are called kernel-mode rootkits. In general kernel-mode rootkits are harder to detect than user-mode rootkits.

This report will lay emphasis on a special kernel-mode rootkit for the Linux operating system. There are different techniques for implementing a rootkit by manipulating the kernel, but most of the time there are also proper countermeasures to detect and remove them from the infected memory. There is however one kind of rootkit that has a history of being undetectable - the debug register rootkit.

---

1 In the chapter “The Linux Kernel” the kernel will be discussed further.
1.1 Related work

As in every other malware, the various rootkit techniques used to compromise computer systems change over time. The popular techniques and exploits that crackers\(^2\) take advantage of get unusable once the developers of the security companies find a way to fix the exploit. When that happens, the crackers find new exploits, thus setting new trends for the rest of the crackers to follow. The arms race between the security companies and the crackers has been going on for decades. Today some security companies publish monthly, quarterly and yearly reports so that the ordinary computer users as well as security experts can keep up with the current trend. The so called white hats\(^3\) may keep their knowledge up to date by reading the security reports, but the black hats have corresponding sources of their own for publishing and discussing new and old exploits. One of these websites is Phrack Magazine [2], an online magazine founded in 1985 that specializes in topics regarding hacking. Over the years, exploits and techniques have been found, published and discussed on Phrack’s website, where users may participate through the comments field for each article. A group of articles is published in every issue, written by one or several of Phrack’s authors. In April 2008, there was a release of a new issue that contained an article [3] that presented a new technique written for computer processors that use the IA-32 (Intel Architecture, 32-bit) architecture.

The article claimed that total stealth can be achieved in any system using the IA-32 architecture, by making use of the debug registers that are unique for IA-32. The debug registers will be introduced more thoroughly in later sections. The article also stated that there is no way of detecting this kind of threat: “Being a fundamental processor feature, means it can be used on ANY operating system running on IA-32 and also, there is no way of detecting or protecting against it, even if it is not 0day anymore.” [3].

According to the author, by the time of writing, the technique had been used in the underground for more than eight years, making it more than 12 years now, should his assumption be correct. The code examples in the article were written for the Linux operating system, but as long as the processor architecture is IA-32, the principles are not limited to an operating system.

Later that year, in September 2008, a post [4] appeared on seclist’s website, in a section called Daily Dave Mailing List. The author released an open source rootkit that was based on the article from Phrack and was free for anybody to download and execute on any Linux system using a 2.6 kernel version. When the author moved the files to the company Immunity [5] a couple of days later, it immediately gained attention from computer news sites, such as The Register [6] and Linux Journal [7].

Three years later, in April 2011, a blog post [8] from one of the researchers of the security company McAfee reported that a rootkit that used the debug registers in a similar manner was encountered. This was the first known encounter of a rootkit on a Windows system that used the debug registers in a malicious way. An odd behavior of the rootkit had caught their attention when it was analyzed. They saw a hook\(^4\) in the memory space, but when they tried to read the memory address, they couldn’t because the memory they read was not the actual

\(^2\) Cracker – short for criminal hacker.
\(^3\) Security experts and ethical hackers are often referred to as “white hats”. Naturally, the crackers are called “black hats”.
\(^4\) A way of intercepting function calls. Further discussion of hooks is presented in later sections.
address. Based on this behavior, the term “memory forging” was coined by the author. This discovery also drew the attention of the computer security websites. At that time, two unrelated sources wrote about the matter: Tech Republic [9] and security expert Bruce Schneier [10], with both authors referring to the technique as “memory forging”.

Currently there is no information about any software or company that manages to detect and remove rootkits of this kind. The researcher at McAfee states: “At McAfee Labs we will continue to investigate and provide protection against such threats.” [8]. Nevertheless it is not clear what kind of protection it is that McAfee provides, or which systems it concerns. It could be preventive protection as well as detection. It could be protection based on signature recognition or heuristic algorithms.

The fact that there may be an ultimate rootkit out there that no one can detect, and the additional fact that it has been out in the open for 12 years motivates an investigation to be able to verify or reject the threat.

### 1.2 Aims and objectives

The research results from McAfee [8] seem to investigate how the rootkit works, but no information has been reported from any major security company about countermeasures for this threat. The information written about the matter is limited. There seems to be a lack of literature on the subject, apart from the originating article from Phrack [3], and some various blog- and forum posts. There may actually be a rootkit technique out in the open that allows a rootkit to operate unchallenged in millions of systems.

The purpose of this report is to once more bring this problem to the surface and investigate which security companies managed to keep up with this threat. Furthermore, this report should inform about some of the common algorithms used in standard- and debug register rootkits. The threat debug registers is relatively new and needs to be observed until it is no longer considered a threat.

The goal is to see if and how well the most popular malware removal tools deal with debug register rootkits. This will determine if Linux users still have to be aware of this threat, or if it has been taken care of already.

### 1.3 Research questions

The research questions that this report will answer are the following:

1. How many of the chosen rootkit/anti-virus detection software have implemented a detection algorithm against the debug register rootkits?
2. Can the Linux operating system be declared safe from this kind of threats?

By answering the questions above it can be determined if Linux is still vulnerable against debug register rootkits.
1.4 Hypothesis

The hypothesis regards steps one and two in the research question list above.

Hypothesis #1: *None* of the detection software is able to detect the rootkits.
Hypothesis #2: The Linux system *cannot* be declared safe from debug register rootkits.

1.5 Methodology

For the background study of the report, mainly electronic sources were used (such as online articles and publications, see references for a complete listing) due to the lack of published books on the subject. In addition, source code to different programs has been studied as well, in preparation for the practical moments. The research started out with various blog- and news posts to better understand the basic concepts. Then, information from more technical sources was studied to get a deeper understanding. Finally, the source code [4] used was read and at some parts modified to adjust it so that it fulfills the criteria of the project.

The method used for answering the research questions will be to compare a single debug register rootkit that only uses the debug registers for two breakpoints, making it fully available for tools to check for integrity modification, against a debug register rootkit with three breakpoints that theoretically should be impossible to detect. Both of the rootkits will use standard hooking mechanisms, and they will both be built on the same code. They will be run against several popular rootkit removal tools on a Linux system that is built on a 2.6 kernel version. If the results vary between the rootkits then a difference between them has been demonstrated, and some new conclusions may be reached about how and why.

For conducting the research, several popular\(^5\) Linux rootkit/anti-virus removal tools will be downloaded, installed and tested against the rootkit files.

1.6 Disposition

Before narrowing down to the actual test, the Background chapter will provide an introduction for the reader to the Linux operating system, the kernel, the debugging mechanism of IA-32, an introduction to hooking, and finally an analysis of the source code for the open source rootkit from Immunity [4].

The Experiment chapter will discuss the preparations for the test and which tools and operating system are being used in the test. Furthermore a motivation to why they were being chosen will be discussed. In conclusion, the actual performance of the test will be conducted. The Results chapter will present all the results from the test. The Discussion chapter will discuss the results. The Conclusions and Future Work chapter will discuss the final conclusions and the future work in this subject. The references will be listed in the end of the report.

\(^5\) A program’s popularity will be determined based on search engine hits and occurrences in online articles/forums.
1.7 Conventions

All source code and command examples provided in the report will be written using the Consolas font. Assembly source code uses the AT&T notation.

2 Background

The purpose of this chapter is to provide some basic knowledge about the parts that is necessary to have in order to understand the Experiment part that is presented in the next chapter. First and foremost, this chapter presents a brief introduction to the relevant parts of the Linux kernel. The rootkit that has been studied in this project operates in the kernel, and therefore a chapter about it has been included. This includes interrupts, the IDT table, system calls and kernel modules. Next is the Debug registers chapter, covering the debug registers that the rootkit makes use of and that are the main part of the debug register rootkits. This is followed by an introduction to hooking, a technique used by many rootkits, including the rootkit that has been examined. Last a chapter about the actual rootkit is included to cover all the necessary parts of information about the rootkit before going into the Experiment chapter. The rootkit chapter summarizes all the previous parts covered in the Background chapter by putting them into context.

2.1 The Linux Kernel

2.1.1 Introduction

First of all, it is important to understand what a kernel actually is, and what it does. How does kernel space differ from the user space? The kernel functions acts as a layer between the hardware and the user space. They decide which applications from user space get access to the hardware. In order for an application to e.g. open a file, it has to pass the request to the kernel. The kernel is able to perform many operations: open files, create processes, threads, execute programs and many other things that are considered privileged. The kernel’s place in the operating system is best described by a hierarchy diagram. The technology corporation IBM has a diagram for describing the architecture of the Linux operating system, which can be seen below in Fig. 1 [11].

![Fig. 1. The Linux architecture](image-url)
In the diagram, the operating system is split into two main parts: the user space and the kernel space. There is actually a third part (the hardware platform) in the diagram, but it is not a part of the operating system. In the user space, the user applications are executed. In order for a user space application to request resources from the kernel, it has to pass it by the System Call Interface (SCI). When requesting the attention of the SCI, a software interrupt has to be generated by the program first.

2.1.2 The Intel X86 protection rings

The different privilege layers of the Intel X86/IA-32 processor architecture are often described as different “protection rings”, or just “rings”. The most privileged space in the system is the kernel, so it is located in Ring 0. The Ring 1 and 2 is for device drivers, and the outmost ring, Ring 3, is for the applications in user space. It is best described by an onion diagram as seen below in Fig. 2. The most important rings are Ring 0 and Ring 3.

![Fig. 2. Onion diagram of the X86 protection rings](image)

This privilege mode is a part of the processor, thus any system, e.g. Linux or Windows using an X86-processor has the same protection structure.

2.1.3 Interrupts and exceptions

An interrupt may be described as a way for software or hardware to get the attention of the CPU (Central Processing Unit). It is executed by sending a signal to the CPU, which in turn processes the signal and acts accordingly to what the signal means. There are two types of interrupts: hardware- and software interrupts. Sometimes, also the word exception is used. An exception may be defined as:

“Exceptions belong to a special type of software interrupts. They are generated by the processor itself whenever some unexpected critical event occurs.” [12]

The most relevant interrupt in this context is the software interrupt. It basically means that the signal sent to the CPU comes from an application in user space. When the CPU is receiving the signal from an application, the right interrupt handler is determined. Each signal has an interrupt handler, a function to handle what the calling program wants to do. The CPU
executes the interrupt handler, and then proceeds with the tasks that it was performing before the interrupt.

The addresses to all the interrupt handlers are stored in the *IDT* (Interrupt Descriptor Table)\(^6\). An illustration of the interaction between the different parts is shown below in Fig. 3.

![Diagram of system interrupt](image)

**Fig. 3.** A software interrupt for a system call (INT 80)

In the diagram, an application is passing a request for a system call; it might for example want to open a file or create a child process. The application sends the request to the C-library, which in turn invokes a software interrupt for the CPU. The signal sent to the CPU indicates that a system call\(^7\) is requested, and the interrupt handler for system calls is located in the IDT at address 0x80, therefore the interrupt handler is called INT 80. The interrupt handler is invoked, the result is passed back to user space, and the CPU continues with its tasks.

2.1.4 The Interrupt Descriptor Table

The Interrupt Descriptor Table (IDT) is a structure that holds interrupts. It is basically a table filled with pointers (addresses) to all the interrupt handlers. The IDT contains 256 [13] entries which are filled out with pointers when the system boots. In Fig. 4 some of the entries in the IDT are presented.

---

\(^6\) The IDT table is covered in the IDT section.

\(^7\) A system call is a request for kernel resources by an application. System calls are being covered in the System call table section.
Some of the IDT entries

Pay attention to the 80th entry in the diagram. That is the pointer to the INT 80 (System call) which was used in the example (See Fig. 3) before. There are methods that software developers may use for reading as well as writing to the IDT. Those methods are called SIDT (Store IDT) and LIDT (Load IDT), respectively. One might think that the SIDT would be for copying data into the IDT, but it actually means that it copies the IDT to the destination operand.

The assembler command below is a short demonstration of how to store the IDT in the ESP (Extended Stack Pointer) register.

```
sidt %esp
```

When calling SIDT or LIDT, the actual IDT is not returned, but merely a pointer to it. This is called the IDTR (IDT Register).

### 2.1.5 The system call table

A system call serves as an interface between the user space and kernel space. When a system call is invoked by the C-library, there is a switch from user mode to kernel mode. As already shown, the interrupt handler for a system call lies in entry #80 in the IDT. The INT 80 is invoked, and it then calls the function `system_call`. In the kernel, there exists another table: the system call table. This table is similar to the IDT, though the system call table stores the addresses to the functions that provide the kernel operations. The function `system_call` finds the right system call in the table and executes it.

Some examples of names and tasks [14] of system calls may be found in Table 1 below.

<table>
<thead>
<tr>
<th>Sys call #</th>
<th>Name</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td><code>sys_fork</code></td>
<td>Create child process</td>
</tr>
<tr>
<td>3</td>
<td><code>sys_read</code></td>
<td>Read from a file descriptor</td>
</tr>
<tr>
<td>11</td>
<td><code>sys_execve</code></td>
<td>Execute program</td>
</tr>
</tbody>
</table>

Table 1. Examples of system calls in the system call table

This is Linux way of letting less privileged applications in Ring 3 execute operations in Ring 0.
2.1.6 Kernel modules

In the previous chapters it has been shown that an application may use system calls to use the privileged functions of the kernel. But is it possible for users to write their own parts of programs for the kernel? Actually, it is possible through kernel modules. By using kernel modules, it is able to extend the functionality of the kernel itself. Kernel modules are nothing more than ordinary programs that may be compiled and inserted from user space directly into the kernel. The insertion process does not require a recompilation of the kernel or a system reboot.

The make utility and makefiles

Kernel modules have some dissimilarity from ordinary programs in that they have to be compiled in a special manner, using special libraries. A common way of doing this in Linux systems is with a makefile. A makefile is an ordinary file in the system that is named Makefile, and contains instructions for building files. It is basically used for automating the building process. A makefile is used together with the Linux make [15] utility to build ordinary programs and kernel modules. The code below shows an example of how some of the contents of a makefile can be structured.

```make
# Compiling source files
main.o: main.c main.h
    cc -c main.c

test.o: test.c defs.h
    cc -c test.c
```

Line by line the code may be described as follows:

1. The object file main.o is dependent on the files main.c and main.h.
2. Compile main.h. This generates main.o.
3. The object file test.o is dependent on the files test.c and defs.h.
4. Compile test.c. This generates test.o.

The code structure above is relatively simple to understand, but the kernel build code uses libraries from the operating system that is necessary to include. A short example of this is shown below.

```bash
obj-m := Kernel.o
all:
    $(MAKE) -C /lib/modules/`uname -r`/build M=`pwd` modules
```
This build will generate a file with a .ko-extension, while a build from the previous example only would generate object files with .o-extensions. The different extensions are used simply to distinguish between the generated files. The use of the Make-utility simplifies and automatizes the building process. The user is not even required to specify the filename if the makefile lies within the current directory. For example:

```
test@test-desktop:~/Desktop$ make
```

The make command will find the makefile in the Desktop directory and build the program based on the defined rules in the makefile.

**Insertion/removal of modules into the kernel**

When the kernel module is compiled, the user only has to insert it into the kernel. As mentioned before, this is done directly at runtime and without recompilation of the kernel. This is made by the `insmod` command. This command requires root privileges to execute. The file to be inserted should be a kernel object (.ko-extension) file. The next example shows an insertion of the test kernel module (which in this case lies in the Desktop directory) in a Linux system:

```
test@test-desktop:~/Desktop$ sudo insmod ./test.ko
```

The user will be prompted for the password, and then the kernel is inserted. The module may be viewed in `/proc/modules`, where every module in memory is listed.

Finally, the inserted module may be removed by executing the `rmmod` command, which also needs root privileges:

```
test@test-desktop:~/Desktop$ sudo rmmod test
```

The module should now be detached from the kernel, and should not be listed anymore in `/proc/modules`.

### 2.1.7 Summary

Some of the most important parts of the Linux kernel have been covered in the sections above, but not all parts of the kernel. Only the parts relevant of this project have been included. The information about the kernel will be put to use in later chapters. Next chapter covers the debug registers, which is a processor-specific feature and is consequently not included here in the Linux kernel chapter.

---

8 Root is a common name for the superuser account on Linux and many other UNIX-like operating systems.
2.2 The debug registers

2.2.1 Introduction

The IA-32 processor has support for debugging operations [16]. There are special registers that the processor provides for this purpose; they are called the debug registers (DR). There are eight debug registers, ranging from zero to seven; they are named DR0-DR7. The registers allow a developer to store addresses from memory locations in them. When that is done, a breakpoint is created. A breakpoint causes the program, once run, to halt the execution and lets the developer inspect the current values of variables and registers.

Each time the processor is accessing a memory location where a breakpoint is set, a debug exception is generated. Looking back to the Fig. 4 in the IDT chapter, the diagram showed that the interrupt handler for debug exceptions is located in the second entry (INT 1) in the IDT. Every time a debug breakpoint is located, INT 1 is executed from the kernel.

In the Linux system, the INT 1 calls a function named do_debug(), which takes care of the necessary operations for debugging. In Fig. 5, this calling order is illustrated. The figure resembles Fig. 4, only with a different interrupt.

The debug functionality is not identical for all debug registers. The similarities and differences between the functionality of specific debug registers are described in the following subsections, which are based on the Intel [16] documentation.

2.2.2 DR0-DR3

The first registers, DR0-DR3 store the linear addresses of the breakpoints. This means that the maximum number of breakpoints that can be set is limited to four.
### Setting a breakpoint

Below is a short piece of code written in assembler that sets a breakpoint in the debug register DR0. The EAX register (Extended Accumulator register) contains the address of the instruction that should be added to DR0.

```
movl %eax, %dr0
```

Now a breakpoint is set for the address in EAX. When the processor reads the instruction, the debug exception handler (INT 1) is invoked.

##### 2.2.3 DR4-DR5

These registers were not used in the project, and will not be described in depth. Intel's own description of the registers is: "Debug registers DR4 and DR5 are reserved when debug extensions are enabled (when the DE flag in control register CR4 is set) and attempts to reference the DR4 and DR5 registers cause invalid-opcode exceptions (#UD)." [16].

##### 2.2.4 DR6

The DR6 register is called the status register. It is responsible for reporting about the conditions of the last debug exception that was generated. The register contains a set of flags that provide information to the developer. The Intel Software Developer manual [16] for IA-32 describes the flags in the status register.

**B0-B3**

Indicates (when set) that its associated breakpoint condition was met when a debug exception was generated. These flags are set if the condition described for each breakpoint by the LENn, and R/Wn flags in debug control register DR7 is true. They may or may not be set if the breakpoint is not enabled by the Ln or the Gn flags in register DR7. Therefore on a #DB, a debug handler should check only those B0-B3 bits which correspond to an enabled breakpoint.

**BD (debug register access detected) flag**

Indicates that the next instruction in the instruction stream accesses one of the debug registers (DR0 through DR7). This flag is enabled when the GD (general detect) flag in debug control register DR7 is set.

**BS flag**

Indicates (when set) that the debug exception was triggered by the single-step execution mode (enabled with the TF flag in the EFLAGS register). The single-step mode is the highest-priority debug exception. When the BS flag is set, any of the other debug status bits also may be set.

**BT flag**

Indicates (when set) that the debug exception resulted from a task switch where the T flag (debug trap flag) in the TSS of the target task was set. See Section 7.2.1, “Task-State Segment (TSS),” for the format of a TSS. There is no flag in debug control register DR7 to enable or disable this exception; the T flag of the TSS is the only enabling flag.

The contents of the DR6 register are never cleared by the processor.

---

9 A flag is an indicator (usually a bit) that is set to a binary value (1 or 0) to represent true respectively false.
2.2.5 DR7

The DR7 register is called the *control register*. It is responsible for enabling and disabling breakpoints, as well as setting the conditions for a breakpoint. As with the description of the flags in subsection 2.2.4 DR6, the Intel Software Developer manual [16] is used to describe the flags in the control register.

**L0 through L3 flags**
Enables (when set) the breakpoint condition for the associated breakpoint for the current task. When a breakpoint condition is detected and its associated Ln flag is set, a debug exception is generated. The processor automatically clears these flags on every task switch to avoid unwanted breakpoint conditions in the new task.

**G0 through G3 flags**
Enables (when set) the breakpoint condition for the associated breakpoint for all tasks. When a breakpoint condition is detected and its associated Gn flag is set, a debug exception is generated. The processor does not clear these flags on a task switch, allowing a breakpoint to be enabled for all tasks.

**LE and GE flags**
This feature is not supported in the P6 family processors, later IA-32 processors, and Intel 64 processors. When set, these flags cause the processor to detect the exact instruction that caused a data breakpoint condition. For backward and forward compatibility with other Intel processors, we recommend that the LE and GE flags be set to 1 if exact breakpoints are required.

**GD (general detect enable) flag**
Enables (when set) debugregister protection, which causes a debug exception to be generated prior to any MOV instruction that accesses a debug register. When such a condition is detected, the BD flag in debug status register DR6 is set prior to generating the exception. This condition is provided to support in-circuit emulators. When the emulator needs to access the debug registers, emulator software can set the GD flag to prevent interference from the program currently executing on the processor. The processor clears the GD flag upon entering to the debug exception handler, to allow the handler access to the debug registers.

**R/W0 through R/W3 (read/write) fields**
Specifies the breakpoint condition for the corresponding breakpoint. The DE (debug extensions) flag in control register CR4 determines how the bits in the R/Wn fields are interpreted. When the DE flag is set, the processor interprets bits as follows:

- 00 — Break on instruction execution only.
- 01 — Break on data writes only.
- 10 — Break on I/O reads or writes.
- 11 — Break on data reads or writes but not instruction fetches. When the DE flag is clear, the processor interprets the R/Wn bits the same as for the Intel386™ and Intel486™ processors, which is as follows:

- 00 — Break on instruction execution only.
- 01 — Break on data writes only.
- 10 — Undefined.
- 11 — Break on data reads or writes but not instruction fetches.
LEN0 through LEN3 (Length) fields
Specify the size of the memory location at the address specified in the corresponding breakpoint address register (DR0 through DR3). These fields are interpreted as follows:

- 00 — 1-byte length.
- 01 — 2-byte length.
- 10 — Undefined (or 8 byte length, see note below).
- 11 — 4-byte length.

2.2.6 Summary
The debug register rootkit technique involves most of the debug registers covered above. The BD flag in DR6 and the GD flag in DR7 is of most importance, even if more flags are used in the source code of the Linux Rootkit. The malicious use of them will be discussed further in the Open source Linux rootkit section.

2.3 Hooking
2.3.1 Introduction
A hook refers to the technique of intercepting a message, event or function in a software system. It is often used in rootkits to intercept functions, and manipulate their functionality. Hooking methods are however not only used for rootkits or other malware; they may be used for other purposes as well. Hooking is merely another tool in the programmer’s arsenal for solving different problems. In Fig. 6, an ordinary function call is shown.

![Fig. 6. An ordinary function call from a program](image)

There is nothing wrong with this type of function call. The program calls a function and when the function has executed, it returns. Control is then once again given to the calling program.

In the next illustration, Fig. 7, this is shown. In the figure an ordinary function call is shown as well. Note that the function hook executes before the real function is called.
2.3.2 Hooks and rootkits

A hook function may be used in a program to execute code before the ordinary function is called. A rootkit that uses hooks may hook the function that lists files in the operating system. This is common for hiding files in a system. When this happens, control is passed to the hook function before the actual file-listing function. The rootkit may then simulate the actual function by listing all files, except the files that are desirable to hide. The rootkit itself is usually also among those hidden files. As seen in the second call in Fig. 7, it is important for a rootkit to pass control back to the file-listing function, once it is done. If control is not passed back, the function will stop working because it is never called.

2.4 The Open Source Linux 2.6 Rootkit

2.4.1 Introduction

The Open Source Linux 2.6 Rootkit [4] (OSLR) was found and downloaded from the computer security company Immunity’s website [5] to use as a foundation for the tests. It is a kernel module, thus it is running in Ring 0. The download package includes a Makefile to build the file using make. The OSLR was originally written for a Linux 2.6 kernel, though on 22 July 2011 the Linux kernel version changed [17] from 2.6 to 3.0. Nevertheless the functionality of OSLR should probably not be endangered, judging by the comments of Linus Torvalds\(^\text{10}\) at the time of release: “So what are the big changes? NOTHING. Absolutely nothing”.

The rootkit consists of two files: DR.c and hooktable.h. DR.c handles the debug registers, and hooktable.h handles the hooking procedure of the various system calls. A list of the available hooks is presented further on.

\(^{10}\) The creator of Linux.
2.4.2 The debug register rootkit

The debug register rootkits are not different from the standard rootkits regarding methods of hiding files, processes, etc. The OSLR uses standard hooking mechanisms to accomplish this. The means of hiding itself is however unique; it takes advantage of the attributes of the debug registers. The OSLR gets the address to the system call handler (INT 80), and sets a breakpoint on its address. This means, that every time a system call is executed, a debug exception (INT 1) is generated. As seen in previous chapters, INT 1 passes control to the function do_debug(). The OSLR replaces the do_debug() with its own malicious function: __my_do_debug(). Every time a system call is generated, control is passed to __my_do_debug(), as seen in the next illustration.

![Diagram](image)

As seen in the summary of the Debug registers section, the status- and control registers (DR6 and DR7) have the BD- and GD flag. The BD flag may be used to check if someone is trying to read the debug registers. If someone is going to read the registers, the flag is set. This is checked before someone tries to read them. The GD flag may be used to generate a debug exception (INT 1) prior to any MOV instruction.

The BD and GD registers may be used together maliciously. When they are set, control is passed to INT 1 when someone is trying to read or write to the debug registers. INT 1 calls function do_debug(), and then the debug registers may be read unconditioned. This means that every time someone (e.g. an anti-rootkit scanner) is trying to read the debug registers, control is passed to the OSLR’s __my_do_debug() first.

When __my_do_debug() is called, it checks what caused the exception; was it due to a system call, or did someone try to read or write to the debug registers? __my_do_debug() has total control over the debug registers. The advantage of using the debug registers is that the rootkit gets the control before anything else. What happens in __my_do_debug() depends on what

---

11 MOV is an instruction used in X86 assembler language for copying data from a location to another.
the attacker has decided to implement. An example of an implementation is when the attacker clears the debug registers every time a scanning tool is trying to read them, and re-installs the rootkit a few milliseconds later when the tool has proceeded to scan in other parts of the memory. The tool will then always discover empty debug registers when it tries to scan them. There is a way to forge the memory as well, as seen in the interview with McAfee’s researcher Rachit Mathur [9], who spoke of how the rootkit served contents from another location in memory. This was possibly another implementation of __my_do_debug(). The rootkits OSLR and TDSS.e!rootkit both have in common that they are both using the debug registers to hide.

The next illustrations, Fig. 9 and Fig. 10 will show an example of how the __my_do_debug() may clear the registers before a scanning tool has any chance to read the actual contents of them. The actual OSLR is not implemented this way from the beginning, but it shows a feasible implementation for the attacker to use.

![Diagram of memory forging rootkit behavior](image)

Fig. 9. Step 1 - A scanning tool trying to read a breakpoint when the GD flag is set

In Fig. 9, the scanning tool is trying to read DR0 where the breakpoint for system calls lies. The GD flag in DR7 is set. This results in a debug exception.

![Diagram of memory forging rootkit behavior](image)

Fig. 10. Step 2 - A debug exception occurs and the registers are cleared

Fig. 10 shows what happens when the debug exception is triggered by the scanning tool. This lets __my_do_debug() clear the registers before the scanning tool gets to see the content. When the tool has left the register locations, the rootkit is set up once again. How does a memory forging rootkit’s behavior differ from the implementation explained above? The
memory forging rootkit does not clear the registers, although it will show the user/scanning tool memory contents from another memory location than they initially requested. This way, the actual memory contents are never being read. As seen in the diagram, there are three breakpoints. They will be covered next.

**System call breakpoint (DR0)**

As illustrated in Fig. 9 and Fig. 10, there are three breakpoints set. One is the INT 80, the handler for system calls. This lets the attacker hook every system call. The OSLR has hooked several system call functions. Some of the hook functions are listed below with the return types and parameters left out:

- hook_getdents64 - Read directory entries
- hook_getdents32 - Read directory entries
- hook_execve - Execute programs
- hook_socketcall - Socket system calls
- hook_fork - Create child process
- hook_exit - Terminate the current process
- hook_chdir - Change working directory
- hook_open - Open a file or device
- hook_kill - Send a signal to a process
- hook_getpriority - Get program scheduling priority

The OSLR has the capabilities to hide files, processes, sockets and more. This breakpoint is important as it lets the attacker get control over every system call function. A new hook to a system call function may easily be added in the rootkits source code.

**Sysenter breakpoint (DR1)**

The OSLR sets a breakpoint on SYSENTER as well. SYSENTER is a faster alternative to INT 80 for transitioning from Ring 3 to Ring 0. It was first introduced on the Pentium II processors [18]. The OSLR lets the attacker decide if a breakpoint should be set on SYSENTER. If not, it will only set breakpoints on INT 80 and INT 1.

**Debug exception handler breakpoint (DR2)**

The last breakpoint is set on INT 1. It is important to mention that the OSLR originally does not set a breakpoint on INT 1 when it is downloaded. The OSLR was written for learning purposes, and the actual stealth part was left as an exercise for the reader to implement.

In order for the rootkit to be completely hidden however, a breakpoint on INT 1 is necessary, and that is why there is a breakpoint for INT 1 in Fig. 9 and Fig. 10 as well. The breakpoint on INT 80 and SYSENTER was merely to gain control over the system calls in the kernel, but why is there a breakpoint on the debug exception handler? The debug handler’s function do_debug() is the only part in the operating system that the rootkit modifies. If a tool checks for integrity changes in do_debug(), it will see the modification made by the rootkit. But by placing a breakpoint on INT 1 as well, control is passed to the attacker before this integrity
check can be made. This is the most important breakpoint in order to theoretically make the rootkit completely invisible.

3 Experiment

3.1 Introduction

The experiment is based on different versions of the OSLR. As mentioned before, the OSLR is originally not implemented to use the debug registers in such a way that it allows total stealth. It is merely a basic rootkit that uses debug registers for storing and generating debug exceptions. For total stealth, the third breakpoint has to be set on INT 1, as previously discussed. Therefore the experiment will be performed with two versions of OSLR, where the first version uses two breakpoints and the second version uses three. To show any difference between the basic- and modified rootkit, both have to be tested against the tools, which are covered in the subsection Tools below. The experiment will start out with the basic rootkit, and then the modified will be tested. The results of the experiment will be presented in the next section.

3.2 Tools

The tools chosen to compare the both rootkits were found by searching the Internet. By using Google’s search engine with keywords like “Linux anti-rootkit” and “Linux rootkit scanner”, some of the most popular tools could be discovered and downloaded. One exception was McAfee’s software, it was not chosen by its popularity. It was chosen because McAfee’s discovery of the Windows memory forging rootkit, and their earlier statement [8] that says that they still provide protection against this kind if threats. It is of interest to see if general debug register behavior is included in their detection algorithms. They have an antivirus-scanner for Linux that has been used in the experiment. The tools that were found and used in the experiment are listed below:

- Rootkit Hunter [19]
- chkrootkit [20]
- AIDE [21]
- ClamAV [22]
- McAfee VirusScan Enterprise [23]

The tools can be divided into three different categories: rootkit scanners, file integrity checker tools and antivirus software. The rootkit scanners are software specialized in finding rootkits. They typically do not care for viruses, worms, or other malware which behavior does not resemble the rootkits. The file integrity checker tools takes a “snapshot” of files, directories or the whole system and saves it to a database. The database contains the checksums of the files. Then the database contents are compared to the system contents to see if something has changed since the retrieval of information. The antivirus scanners are despite their category name not limited to just viruses. They often recognize other malware as well, and some software suites have anti-rootkit modules built into the software.
Below are the programs divided in their categories:

**Rootkit scanners**
Rootkit Hunter  
chkrootkit

**File/directory integrity checker tools**
AIDE

**Antivirus software**
ClamAV  
McAfee VirusScan Enterprise

Why are there no programs from the major security software companies in the list? This is mainly because these programs are designed with the Windows operating system in mind. Even if they were installed on a Linux machine, they would still only search for Windows malware. The range of scanning tools for Linux is very limited, since the operating system is less targeted and is considered safer than Windows [24]. The list above should however cover the average Linux user’s range of tools.

### 3.3 The experiment

The experiment will test one tool at a time. They are all divided into subsections as seen below. In the experiment, a rootkit will hide one file and one directory by hooking the `getdents` (Read directory entries) system call. This is considered a malicious action. The rootkit itself is not hidden; hence the malicious module may be found in `proc/modules`. In the Desktop directory on the test system, there is a directory called `Malicious_Files` (See Fig. 11). In Malicious_Files lies a file and a directory named `Hide_Me_File` and `Hide_Me_Folder`, respectively (See Fig. 12). The rootkit will hide every file and directory which name contains the string `Hide_Me`.

![Fig. 11. Directory Malicious_Files on the Desktop](image)
The rootkit is inserted into the kernel in the same manner as seen previously in the subchapter *Kernel modules* from the Background chapter:

```
test@test-desktop:~/Desktop$ sudo insmod ./Rootkit.ko
```

The rootkit is now inserted into the kernel and the file *Hide_Me_File* and the directory *Hide_Me_Folder* are both confirmed to be hidden before any tests begin. Note that the files will be hidden automatically directly at the rootkit insertion. The rootkits will be tested against the tools, starting out with the basic rootkit.

### 3.3.1 Rootkit Hunter

Rootkit Hunter is a rootkit scanner for Linux. On its website [19] some of the features can be found:

- MD5 hash compare
- Look for default files used by rootkits
- Wrong file permissions for binaries
- Look for suspected strings in LKM and KLD modules
- Look for hidden files
- Optional scan within plaintext and binary files

Rootkit Hunter is installed by downloading the files from the official website [19], or by using the following command:

```
test@test-desktop:~/Desktop$ sudo apt-get install rkhunter
```

The version of Rootkit Hunter used in the test is 1.4.0. Rootkit Hunter is then started from the terminal using the following command:

```
test@test-desktop:~/Desktop$ sudo rkhunter --check
```

Some results are presented to the user directly. This may include warnings for some suspicious activities. The verbose results are saved to *var/log/rkhunter.log*. 

![Fig. 12. Hide_Me_File and Hide_Me_Folder](image)
3.3.2 Chkrootkit

chkrootkit (Check Rootkit) is an open source rootkit scanner tool.

Downloading chkrootkit is not possible from the official website [20] because of broken links. chkrootkit is installed by the following command:

```
test@test-desktop:~/Desktop$ sudo apt-get install chkrootkit
```

The scanning is then performed from the terminal by typing:

```
test@test-desktop:~/Desktop$ sudo chkrootkit
```

The results are presented to the user directly, and no log file is used.

3.3.3 AIDE

AIDE is a file integrity checker that creates a database of information about files that the user has specified in a configuration (config) file. In the config file aide.config it is possible for an administrator to define rules for the program. It is possible to check a file’s integrity in several ways, e.g. the file size, permissions and checksum.

AIDE can either be downloaded from the official website [21], or using the following command:

```
test@test-desktop:~/Desktop$ sudo apt-get install aide
```

For this test, the config file (/etc/aide/aide.conf) for AIDE is prepared with some presets. The preset Full will be used in the test. This means that changes in the following attributes will be registered by AIDE:

- Permissions
- User
- Group
- Number of links
- inode
- Size
- Block count
- mtime
- ctime

Together with these attributes, a combination of the following message digest algorithms will be used to create a checksum of the files:

- md5
- sha1
- rmd160
- haval
- gost
The snapshot will be taken of the following directories:

- `usr/src/`
- `/home/test/Desktop/Malicious_Files/`
- `/dev/`
- `/lib/udev/devices/`

The directory `usr/src/` holds the kernel sources and the header files. This is included because the function `do_debug()` is defined there. The directory `/home/test/Desktop/Malicious_Files/` is included because the file `Hide_Me_File` and the directory `Hide_Me_Folder` are both hidden there when the rootkit is inserted. The directory `/dev/` holds the files for the physical and virtual memory, `mem` and `kmem` respectively. The last directory, `/lib/udev/devices/` is included because the directory holds a file that is named `kmem`, just like `/dev/`. The contents of these four folders are most likely to be modified by the rootkit, and that is why they were chosen to be specified in the configuration file. If every directory on the system were specified in the configuration file, the log file would be enormous due to the automatic creation and modification of files that the operating system initiates all the time. There would be too many false positives.

Before the database is created, a few preparations are necessary by the user. First, the file `aide.db.new` (other filenames may be specified in the configuration file) has to be created at the directory `/var/lib/aide`.

```
test@test-desktop:~/Desktop$ sudo touch aide.db.new
```

Next, a snapshot of the current state of the specified directories is created, as can be seen below in the next command. Note that this must be done before the insertion of the rootkit.

```
test@test-desktop:~/Desktop$ sudo aide -c /etc/aide/aide.conf --init
```

The snapshot should now be saved to the file `aide.db.new`. Now, copy `aide.db.new` to a new file, named `aide.db`. Again, the filenames may be different depending on the presets in the configuration file `aide.conf`. Now that a copy exists of the database, `aide.db.new` may be overwritten. Now the rootkit is inserted, and another snapshot is now created using the previous `init-command` (this will overwrite the `aide.db.new`-file). There are now two different snapshots of all the directories: one before the rootkit insertion and one after. Now they are checked for differences with the command below:

```
test@test-desktop:~/Desktop$ sudo aide -c /etc/aide/aide.conf --check
```

AIDE scans for differences between the snapshots and reports them to the user.
3.3.4 ClamAV

ClamAV is an open source antivirus program that may be installed and run on both Linux and Windows. ClamAV only detects malware, it does not remove them.

“ClamAV is an open source (GPL) antivirus engine designed for detecting Trojans, viruses, malware and other malicious threats. It is the de facto standard for mail gateway scanning. It provides a high performance mutli-threaded scanning daemon, command line utilities for on demand file scanning, and an intelligent tool for automatic signature updates.” [22]

ClamAV may be installed via the Terminal:

test@test-desktop:~/$ sudo apt-get install clamav

A scan of the whole system is then performed with the command below. It basically executes ClamAV and tells it to recursively scan every folder and all its subfolders on the system.

test@test-desktop:~/$ clamscan -r /

The results are presented to the user in the Terminal directly after the scan.

3.3.5 McAfee VirusScan Enterprise

Unfortunately, when McAfee VirusScan Enterprise for Linux is downloaded from McAfee’s website and extracted, an error message pops up and tells the user that an unexpected end of file (EOF) has been reached. The error message can be seen below in Fig. 13. The package from the website is corrupted, and no other version exists for Linux kernel version 2.6.

![Fig. 13. Archive Manager error message](image)

The experiment will proceed without any results from this software.
4 Results

The results from the test in the previous chapter are presented in this chapter, and discussed in the next. An overview of the results for each of the tools is presented in Table 2 below.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Registered changes</th>
<th>Rootkit found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rootkit Hunter</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>chkrootkit</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>AIDE</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>ClamAV</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>McAfee VirusScan Enterprise</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 2. Results from the experiment

4.1 Rootkit Hunter

Rootkit Hunter is used to scan both before and after rootkit insertion, in order to look for differences. Both rootkit scans result in warnings about suspicious applications and hidden files on the system. Both of the scans result in the same warnings, and thus none of them is related to the rootkit or its effects on the system.

As Fig. 14 shows at the heading Possible rootkits, the warnings do not relate to any rootkit.
4.2 chkrootkit

chkrootkit does not find any rootkits when it is performing its scan. Some warnings are generated, though the same warnings are generated even without the rootkit inserted.

```
Checking 'asp'... not infected
Checking 'bindshell'... not infected
Checking 'lkm'... chkproc: nothing detected
Checking 'rexecs'... not found
Checking 'sniffer'... lo: not promisc and no packet sniffer sock:
eth0: PACKET SNIFFER(/sbin/dhcclient3[27716])
Checking 'w55800'... not infected
Checking 'wted'... chkwtmp: nothing deleted
Checking 'scalper'... not infected
Checking 'slapper'... not infected
```

Fig. 15. chkrootkit scan result

4.3 AIDE

AIDE did notice some changes in the second snapshot. It indicated that files from the directory `/home/test/Desktop/Malicious_Files` that were previously there now have been removed. This can be seen in Fig.16, which is the result from the comparison. Also, when the directory `/dev/` was added in the configuration file, AIDE discovered changes there as well. When the rootkit module is inserted, the files `/dev/udev` and `/dev/udev/uevent seqnum` are modified.

```
AIDE found differences between database and filesystem!!
Start timestamp: 2012-05-26 12:14:51

Summary:
  Total number of files: 29518
  Added files: 0
  Removed files: 2
  Changed files: 0

------------------------------------------------------------------------
Removed files:
------------------------------------------------------------------------
removed: /home/emil/Desktop/Malicious_Files/Hide.Me.File
removed: /home/emil/Desktop/Malicious_Files/Hide.Me.Folder
```

Fig. 16. The AIDE log output from the snapshot comparison

The rootkit itself is not detected, although the effects of it are.
4.4 ClamAV

ClamAV did not find the rootkit on the system; neither did it generate any warnings about suspicious activity. The scan results are illustrated below in Fig. 17.

```
          SCAN SUMMARY
Known viruses: 1235168
Engine version: 0.95.3
Scanned directories: 16959
Scanned files: 90644
Infected files: 0
Data scanned: 3614.78 MB
Data read: 3868.49 MB (ratio 0.93:1)
Time: 617.604 sec (10 m 17 s)
```

Fig. 17. ClamAV scan results

4.5 McAfee VirusScan Enterprise

No results available. See previous subsection 3.3.5 for more information.

5 Discussion

No software used in the experiment could find the rootkit while it was hiding files. AIDE was the software that came closest to discover the rootkit. When comparing the snapshots that it took of the system before and after the rootkit insertion, it was discovered that the latter snapshot indicated changes in the directory Malicious_Files, where the file Hide_Me_File and directory Hide_Me_Folder was hidden. The scan summary reported that they both had been removed. While this is true, there were no indications that this was due to the work of the rootkit. When a snapshot of a neutral system was compared to a rootkit-infected one, it also reported alterations in the directory /dev/ on the files /dev/.udev and /dev/.udev/uevent seqnum. It is presumed that this was from the rootkit insertion. Given that information, along with the reports of sudden removal of files on the system, the system administrators have to come to their own conclusions about what they are dealing with.

The rootkit scanners (Rootkit Hunter and chkrootkit) that are specialized in rootkit detection did not find any rootkit, or even suspicious activity. Both programs did several signature checks against well-known rootkits. Lastly, they both checked for common rootkit behaviors without any results. These are some of the most popular rootkit removal tools on Linux, and because of that it is of great importance that this software is able to detect threats of this kind. Why they cannot detect the rootkit is still a question, however there are presumably two reasons to why they fail to do so. The first reason is that the tools do not check for integrity in the IDT-table, where INT 1 lies. The do_debug() function is modified, and therefore some integrity modifications should be registered by a tool. The second reason is that the tools omit the debug registers. If they do not scan in those places, the hooks that the rootkit placed will never be discovered.
Unfortunately, McAfee’s software could not be installed on the Linux system. This prevents an investigation about how well their software manages debug register rootkits. The previous statement [8] made by Mathur Rachit that claims that McAfee can provide protection against this threat remains unchallenged for now.

But should there not have been a second test with an extended debug register rootkit that is based on the already tested rootkit? Yes, although since no rootkit removal software could find the rootkit that was first tested on the system, it would not have been possible to determine if there are any differences between them. As shown in previously covered chapters, the main difference between the rootkits would have been that the extended rootkit placed a third breakpoint, instead of only two. As it is now, the third breakpoint is not required to compromise a Linux system that uses the tools from the experiment. Then again, this depends on what the attacker intends to do. This test has used only one malicious action: to hide files from the user on the system. The outcome could have been different if the rootkit used other malicious actions as well. Hiding files is a very common feature for rootkits however, and that is why the experiment has emphasized in this.

As previously discussed, Linux still remains one of the safer operating systems. It is not as attractive as other operating systems to attack, and this is reflected on the poor range of available tools to download for the system. This report has not covered the infection procedure. But given that the attacker already has infected the system, for instance through getting direct physical access to the compromised computer, it is hard to discover any changes without integrity tools. Even then, a solid understanding of the operating system is required to notice any changes.

This section will be summarized by answering the initial research questions stated in subsection 1.3.

1. How many of the chosen rootkit/anti-virus detection software have implemented a detection algorithm against the debug register rootkits?

**Answer:** None of the tested tools have any algorithm that is able to detect debug register rootkits.

2. Can the Linux operating system be declared safe from this kind of threats?

**Answer:** No. At least not from debug registers that hides files and directories from the user.

Both hypotheses that were listed in subsection 1.4 may be declared true.

### 6 Conclusions

The debug register rootkits can still be considered a threat, at least for the Linux operating system. There are too few and simple tools for detecting rootkits, although this may be balanced by the difficulty of infecting the operating system. Intel currently provides the most popular processor architectures [25], where the debug registers reside. Initiatives towards this issue have to come from either Intel or the security industry. If the threat spreads on Windows systems and no one is prepared, we may find ourselves in a critical situation. This
threat needs to be further investigated so that the security industry can be prepared for it. The danger lies within the architecture; it is a built in feature that is available to many operating systems. Coming up with a solution to the problem that does not limit the debug register feature is a great challenge.

How bad the threat really is remains unanswered, but this report has shed some light on the problem and verified its existence. This allows for further research on the subject. One thing can be stated however, the Linux operating system (2.6) cannot be considered safe against such debug register rootkits.

Chances are that this rootkit is still as dangerous today as it was 12 years ago.

7 Future Work

This section presents a short list of suggestions on future work on this subject. Anyone who is interested in improving their knowledge for operating systems, kernels, processors and rootkits may find some projects of interest. They all involve practical moments.

The debug register rootkits do still pose a threat and should not be counted out. There is a great amount of research that still could be conducted within this area. The first Windows debug register rootkit was recently discovered [8], and there might be an increase of them in the near future.

Comparison - A comparison between debug register implementations on the Linux system and the Windows system would certainly be of great interest, as the kernels of the both systems are different.

Windows project - Since the debug registers are a processor-specific feature for the IA-32 architecture, the research is not limited to the Linux or Windows operating systems. There is a greater range of tools for the Windows system than for Linux, and therefore another suggestion would be to perform the same investigation as this project has done, although only on the Windows platform instead of Linux.

Why cannot the tools find it? - The experiment in this project could not determine why the debug register rootkit was not found. Did the tools look in the debug registers at all? Or did they only miss the modification of the do_debug()? These are questions that are of importance and that may be of interest in future projects.

Prevention/detection/response - There is still a need for countermeasures for this kind of threats. Research about preventive methods, detection or response to the treats would contribute to this research area.

Test more tools - The research about the debug register rootkits on the Linux system may also be continued from where this project ended. By finding new ways to detect the rootkit, it will open up a way to finish the implementation of OSLR. It will then be possible to further investigate the capabilities of the debug register rootkit and extend it with new features, such as the previously discussed memory forging technique.
References