Implementation of a State-of-the-Art Security Architecture in the Linux Kernel

Niels Avonds

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdspecialisatie Veilige software

Promotor:
Prof. dr. ir. Frank Piessens

Assessoren:
Dr. ir. L. Desmet
Dr. ir. J. C. Gomez Carranza

Begeleiders:
Ir. P. Agten
R. Strackx

Academiejaar 2012 – 2013
Preface

Before introducing the results of the hard work of the past year, I would like to thank a few people without whom this master’s thesis would not have been possible. First of all, I’d like to thank my supervisors, Pieter Agten and Raoul Strackx, for their continuous help and support, for their quick replies and for everything they’ve taught me over the past year. I would also like to thank my promoter, prof. dr. ir. Frank Piessens, for giving me this opportunity. Next, I’d like to thank Bram Gotink for reading my thesis. Finally, I’d like to thank all of my friends and family, particularly my mother Marleen Cabuy, my brother Yurrit Avonds and my girlfriend Els Eyckmans, for their moral support and encouragement.

Niels Avonds
Contents

Preface i
Abstract iv
List of Figures and Tables v
List of Abbreviations vi
1 Introduction 1
   1.1 Attacker Model 2
   1.2 Self-Protecting Modules 2
   1.3 The Linux Kernel 3
   1.4 Contributions 3
2 Background 7
   2.1 Buffer Overflow Attacks 7
   2.2 Damage Reduction 11
   2.3 Conclusion 14
3 Self-Protecting Modules 15
   3.1 Module Layout 16
   3.2 Operations 17
   3.3 Secure Communication 18
   3.4 Multithreading 20
   3.5 Implementations 20
   3.6 Conclusion 23
4 Privilege Containment 25
   4.1 Impact of a Compromised Module 25
   4.2 Reducing Privileges 27
   4.3 Conclusion 29
5 Implementation 31
   5.1 The Linux Kernel 31
   5.2 New Structures 35
   5.3 New Functionality 37
   5.4 Conflicting System Calls 39
   5.5 Multithreading Issues 42
   5.6 Compiler 44
# Contents

6 Prototype Evaluation ............................................. 45
   6.1 SPEC Benchmark ........................................... 45
   6.2 Microbenchmarks .......................................... 46
   6.3 Macrobenchmarks ......................................... 48
   6.4 Security Evaluation ...................................... 50
   6.5 Conclusion ................................................. 51

7 Applications ......................................................... 53
   7.1 Securing Critical Portions ............................... 53
   7.2 Isolating Vulnerable Portions ............................ 55
   7.3 Completely Modularised Applications ..................... 57
   7.4 In-Process Plugins ....................................... 58
   7.5 Trust Assessment Module .................................. 59
   7.6 Summary .................................................. 62

8 Conclusion .......................................................... 63
   8.1 Challenges .................................................. 63
   8.2 Future Work ............................................... 64
   8.3 Contributions ............................................. 65

A Paper ................................................................. 69
B Article (Dutch) ...................................................... 85
Bibliography ........................................................ 93
Abstract

Estimates show that around 30% of the computers worldwide is infected with malware. Malware spreads quickly and automatically by exploiting low-level vulnerabilities in user-level applications, such as buffer overflows. This type of attack leads to the ability to execute arbitrary instructions in a process’s address space. The attacker not only gains full access to the application’s secrets, but is also able to request services of the operating system with the same access rights as the user running the exploited process.

Existing countermeasures focus on prevention of specific attack vectors, but crafty attackers continuously develop new exploits. Consequently, recent research focuses on damage reduction instead, assuming that successful attacks are inevitable. In one research track, security-critical functions are protected by placing them in separate modules that are fully isolated from the rest of the application.

This master’s thesis introduces modifications to the Linux kernel that allow applications to be divided into such protected modules. Rather than focusing on the isolation of security-critical functions alone, the Linux kernel modifications enable completely modularised applications. By guaranteeing that an attacker able to compromise a single module cannot extend his reach to the other modules, his power over the rest of the application and its secrets is limited. The attacker’s power with respect to the operating system is reduced significantly as well, by selectively disabling system calls for each module.

In this thesis, the design of these modifications is discussed and a prototype implementation is described and evaluated. The modified kernel retains compatibility with existing hardware and software. Legacy applications and protected applications thus run side by side, on top of the same operating system kernel. The performance of legacy applications is shown to be unaffected while protected applications show considerable security improvements with an acceptable performance overhead.
List of Figures and Tables

List of Figures

2.1 A vulnerable call stack, before and after the attack. ................. 8
2.2 Stack canaries protect the return address. ......................... 9
3.1 Layout of a process containing an SPM. ......................... 16
3.2 The Fides prototype. Partially accessible memory regions are shown as hatched regions. ........................................... 21
5.1 Virtual memory in the Linux kernel. ............................ 33
5.2 New structures in the Linux kernel. ............................ 37
5.3 Forking a thread in the Linux kernel. .......................... 42
6.1 Relative results of the SPECInt 2006 benchmark. ................. 46
6.2 HTTPS-Server performance. .................................. 49
6.3 gzip performance. ........................................ 50
7.1 Protecting small security-critical portions. .................... 54
7.2 Isolating vulnerable portions of applications. ................... 56
7.3 Securing applications by complete modularisation. ............. 57
7.4 Sandboxing in-process plugins. ................................ 59
7.5 A Trust Assessment Module protected using Self-Protecting Modules. . 60

List of Tables

3.1 The enforced memory access control model. ....................... 17
3.2 Operations in the different supporting systems. .................. 19
6.1 System call overhead. ....................................... 47
6.2 Entering an SPM. ........................................ 47
6.3 Module access overhead. ...................................... 48
7.1 Securing applications using Self-Protecting Modules. ........... 62
8.1 Overhead of memory mapping checks on SPM switches. .......... 64
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>Certificate Authority</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>GNU</td>
<td>GNU’s Not Unix</td>
</tr>
<tr>
<td>GPL</td>
<td>General Public License</td>
</tr>
<tr>
<td>gzip</td>
<td>GNU zip</td>
</tr>
<tr>
<td>IPI</td>
<td>Inter-Processor Interrupt</td>
</tr>
<tr>
<td>MMP</td>
<td>Mondriaan Memory Protection</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PAL</td>
<td>Piece of Application Logic</td>
</tr>
<tr>
<td>PCR</td>
<td>Platform Configuration Register</td>
</tr>
<tr>
<td>SPEC</td>
<td>Standard Performance Evaluation Corporation</td>
</tr>
<tr>
<td>SPM</td>
<td>Self-Protecting Module</td>
</tr>
<tr>
<td>SVM</td>
<td>Secure Virtual Machine</td>
</tr>
<tr>
<td>TAM</td>
<td>Trust Assessment Module</td>
</tr>
<tr>
<td>TCB</td>
<td>Trusted Computing Base</td>
</tr>
<tr>
<td>TPM</td>
<td>Trusted Platform Module</td>
</tr>
<tr>
<td>TXT</td>
<td>Trusted eXecution Technology</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>VMM</td>
<td>Virtual Machine Monitor</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Aside from social engineering techniques, malware mostly spreads by means of exploitation of application vulnerabilities[23]. Bugs in user-level applications allow an attacker to take control over the computer and execute arbitrary code, with all the permissions of the exploited application[33].

Despite numerous countermeasures[11, 25] to prevent existing attacks, new attack techniques continue to appear[41, 53], working around those countermeasures. Consequently, exploitation of application-level vulnerabilities remains a realistic threat. PandaLabs even expects software vulnerabilities to be the main target of cyber criminals in 2013[36].

As 30% of the computers worldwide is infected with malware[36, 5], the implemented preventive countermeasures are clearly unsuccessful. Recent research thus focuses more on reducing the power of an attacker, assuming that a successful attack will occur at some point. A first research track[17, 52, 28, 60] introduces systems that isolate security-critical code or data, reducing the attacker’s power over the compromised application. Other research[16, 21] is more concerned with limiting the attacker’s access to the operating system (OS).

Existing work[52, 51] introduces the concept of fully isolated software modules, called Self-Protecting Modules (SPMs). These modules are used to isolate security-critical functionality of applications. An attacker able to compromise a single module, has limited access to the rest of the application.

This master’s thesis employs and expands the power of these SPMs to reduce an attacker’s influence on the underlying operating system as well. Each module’s access to the operating system is limited according to the principle of least privilege. An attacker able to compromise a single module will be limited in the same way the module is.

This chapter starts by describing the attacker model in section 1.1. Next, section 1.2 introduces the concept of Self-Protecting Modules in a little more detail. Section 1.3 briefly introduces the Linux kernel. Finally, section 1.4 discusses the contributions made by this thesis and gives an overview of the different chapters.
1. Introduction

1.1 Attacker Model

The attacker in this master’s thesis has one very powerful capability: the ability to execute arbitrary instructions inside the user-level application’s memory space. This kind of access can be achieved by exploiting security-critical bugs in the application[33, 11]. Note that the focus is on bugs in user-level applications. The operating system kernel is assumed to be secure.

Because the attacker can execute arbitrary instructions, he can also request the execution of arbitrary system calls from the kernel[25]. Attackers often use these system calls to spawn a shell under their control, enabling them to execute any shell command with the privileges of the user running the exploited application.

Moreover, the attacker is also able to deploy his own modules in the application and request their protection. Consequently, modules requesting protection cannot be trusted. Thus, each Self-Protecting Module must be isolated from code running in the modules of other, possibly malicious stakeholders as well.

Since remote exploitation of computer systems is much more common than physical attacks, the attacker in this master’s thesis is assumed to be a remote attacker. He is thus not able to perform a physical attack on the system. An attacker with physical access could, for example, place a probe on the memory bus, reading the contents of all memory transferred to the CPU.

Finally, the standard Dolev-Yao model[10] is used regarding the cryptographic primitives. An attacker is able to observe, duplicate and modify the messages, but the cryptographic primitives cannot be broken.

1.2 Self-Protecting Modules

The basic idea of Self-Protecting Modules is to securely isolate a portion of an application’s code and data from the rest of the application, while both remain in the same virtual memory space. As an example, consider an application that uses cryptography to securely transmit information over a network. If an attacker is able to execute arbitrary instructions in this process’s memory, he can expose the cryptographic key. By placing this key and the encryption code that uses it in an isolated module, the key never has to leave the module. Consequently, an attacker able to execute arbitrary code in this process, but not in this module, has no access to the key.

An SPM consists of two sections: a Secret section and a Public section. The Secret section contains the confidential data that belongs to the module, the data that should be shielded from code running outside of the module. The Public section contains all of the module’s code, as well as data that should only be integrity protected. As the name suggests, this section is readable from anywhere in the application, enabling authentication and secure communication between protected modules.

Previous work already described two ways of implementing support for Self-Protecting Modules: a hardware implementation[52] and an implementation in a
1.3. The Linux Kernel

hypervisor[51]. In this thesis, however, one of the goals is to remain compatible with existing hardware, ruling out hardware modifications. The hypervisor implementation on the other hand has serious performance drawbacks, making it unsuitable for completely dividing applications in modules. Consequently, this thesis implements support for Self-Protecting Modules in an operating system kernel, more specifically in the Linux kernel. This has the additional advantage that existing system calls can still be used, which makes it significantly easier to port existing applications to protected applications.

1.3 The Linux Kernel

The Linux kernel is an open source operating system kernel. Original development was started by Linus Torvalds in 1991[42]. Today, the kernel powers all Linux distributions, including Ubuntu. Moreover, Google’s Android operating system, the global market leading mobile phone OS[24, 26], is also based on the Linux kernel. The kernel is released under the GNU General Public License (GPL) Version 2.

The source code of the Linux kernel is freely available on the Internet. Anyone can download it and modify it. Since the kernel includes most if not all features expected of a modern general purpose operating system, and because it is used on actual production systems, it is an interesting basis for operating system research. For this reason, the implementation in this thesis was carried out in the Linux kernel.

1.4 Contributions

The original goal of this master’s thesis was to implement and evaluate support for Self-Protecting Modules in the Linux kernel. Additionally, the idea of Self-Protecting Modules was expanded to limit an attacker’s power over the underlying OS and an existing compiler was adapted to ease the creation of SPMs. More specifically, the following contributions are made in this thesis:

• The existing mechanism of Self-Protecting Modules is expanded. The access a module has to the operating system is limited according to the principle of least privilege. Because the modules are fully isolated from each other, an attacker able to compromise a single module will not be able to compromise additional modules without exploiting vulnerabilities in these modules. Consequently, the privileges an attacker has over the underlying operating system are reduced to the privileges of the compromised module.

• Support for Self-Protecting Modules and the additional privilege reduction is implemented in the Linux kernel. New system calls are introduced so applications can request protection of part of their memory space.

• For the hypervisor implementation, a compiler exists that compiles standard C code to SPMs. This compiler is adapted to create SPMs that run on top of the Linux kernel implementation as well.
1. Introduction

- New types of applications enabled by the privilege reduction mechanism and the implementation in the Linux kernel are discussed. Full modularisation of applications was previously impossible due to the hypervisor implementation pausing the legacy operating system while executing code inside an SPM.

- The implementation in the Linux kernel is shown to be fully backwards compatible with existing applications. Moreover, the performance of legacy applications not using the modularisation shouldn’t be affected. Macrobenchmarks show that the performance overhead introduced for applications that use Self-Protecting Modules is limited to 13%.

Each of these contributions will be introduced in the remainder of this master’s thesis. The following list gives an overview of the chapters:

Chapter 2: Background This chapter discusses the minimum required background knowledge. This includes an example of an attack where arbitrary instructions can be executed and a discussion of existing countermeasures. Moreover, existing techniques for limiting the power of an attacker after a successful exploit are described.

Chapter 3: Self-Protecting Modules This chapter discusses the mechanism of Self-Protecting Modules in more detail, describing the program counter dependent access control model, the operations implemented by the supporting system and the mechanism for secure communication between modules. The different ways of implementing support for Self-Protecting Modules are compared, listing their advantages and disadvantages.

Chapter 4: Privilege Containment This is the first chapter that introduces new work. Here, the concept of Self-Protecting Modules is expanded by reducing the privileges each module has over the underlying operating system.

Chapter 5: Implementation The technical aspects of this thesis are discussed in this chapter. First, the relevant parts of the Linux kernel are discussed, including process and memory management, system calls and scheduling. Next, the newly introduced structures and system calls are described in depth. This chapter also briefly describes the compiler implementation.

Chapter 6: Prototype Evaluation This chapter presents an evaluation of the prototype. The performance is evaluated using a system-wide stress test, a set of microbenchmarks and two macrobenchmarks. The final part of this chapter discusses the security of the Linux kernel implementation.

Chapter 7: Applications In this chapter, a general discussion is given of how applications can benefit from the mechanism of Self-Protecting Modules and the expansions introduced in this master’s thesis. For each application, an overview of its memory layout is used to illustrate the discussion and the most suitable supporting system (hardware, hypervisor or kernel) is presented.
Chapter 8: Conclusion  This chapter summarises the results of this master’s thesis. Special attention is given to the challenges encountered during the thesis. An overview of possible improvements and extensions is given as well.
Chapter 2

Background

The field of software security consists of many different aspects. Chapter 1 already briefly situated the problem addressed in this master’s thesis and introduced the concept of Self-Protecting Modules. Section 1.1 introduced the attacker model. This attacker model assumes an attacker able to execute arbitrary code inside an application’s address space. However, no technical details about how such access is obtained were given.

In this chapter, the relevant background in the field of software security is discussed. Section 2.1 discusses how an attacker can compromise a user-level process to the point where he is able to execute arbitrary instructions, filling in the technical gaps left in the discussion of the attacker model. This section also describes some existing countermeasures. In section 2.2, research focused on damage reduction is discussed. Finally, section 2.3 concludes this chapter.

2.1 Buffer Overflow Attacks

The mechanism of Self-Protecting Modules was designed to guarantee the confidentiality of secret data, even if the process containing such data becomes compromised. This means the data is inaccessible to an attacker able to execute arbitrary instructions inside the process’s address space. This section describes buffer overflow attacks, one way for an attacker to gain such access. The technical details of this attack are discussed in more detail in section 2.1.1. Since these attacks are so common[8], a number of countermeasures have been developed. These are discussed in section 2.1.2.

2.1.1 Technical Details

There are two requirements for an application to be vulnerable to a buffer overflow attack[49]. First of all, the application should contain a buffer. In C, this is simply an array. Such a buffer has a predefined length in the process’s memory. The content of the buffer should be under the control of the user. For example, the buffer could be used to temporarily store user input. The vulnerability results from an unsafe copy operation into the buffer, one where no bounds checking is performed. An
2. Background

attacker that enters more bytes than the number of bytes reserved for the buffer is then able to write past the end of the buffer, overwriting other parts of the process’s memory.

The second requirement is the presence of control data in the vicinity of the buffer. Examples of control data include function pointers or function return addresses. An attacker is able to inject code in the buffer, since he has full control over its contents. This is often referred to as the attacker’s shellcode. By changing the nearby control data to the address of the buffer, the attacker causes the application to execute his injected code at some point in the future.

A number of different types of buffer overflow attacks exist. In its simplest form, the attack consists of writing past the bounds of a buffer allocated on the stack, overwriting the return address. This type of buffer overflow attack is shown in Figure 2.1. Figure 2.1a shows what the stack looks like in a normal situation. When the buffer is overflowed by an attacker and the return address is changed to point back to the buffer, the stack will look like the one shown in Figure 2.1b. Note that no countermeasures are shown in these drawings.

![Figure 2.1](image)

(a) Before the attack.  (b) After the attack.

**Figure 2.1:** A vulnerable call stack, before and after the attack.
2.1.2 Countermeasures

Because of the frequent occurrence of buffer overflow vulnerabilities, a great amount of research has been conducted to protect against this type of attack. In this section, the most well-known countermeasures are discussed[11, 25].

**Stack Canaries** Attackers willing to use the exploit described in section 2.1.1 need to overwrite the return address on the stack. This crucial bit of control flow information makes the processor jump to the injected code after the current function finishes. The basic idea of introducing stack canaries is to protect the return address from modifications. After overflowing the buffer and overwriting the return address, all information on the stack located in between the buffer and the return address is destroyed as well. To protect the return address, a random value, or canary, is placed in between the return address and the buffer. Figure 2.2 illustrates this idea. Right before the function returns by jumping to the stored return address, the canary on the stack is compared to the original one. If the canary has been modified, a buffer has been overflowed, and the program will be terminated.[9]

![Figure 2.2: Stack canaries protect the return address.](image)

The problem with stack canaries is that they only protect against stack-based buffer overflows that use the return address to take over the control flow. A similar attack is possible if a regular function pointer is compromised, but stack canaries do not protect these. Other memory corruption attacks are also not mitigated by the introduction of stack canaries.

**Address Space Layout Randomisation (ASLR)**[54, 47] To successfully execute injected code, an attacker must overwrite a function pointer or a return address, changing it to point to his own code. This second countermeasure is designed to make it significantly harder for an attacker to figure out the location of the injected code. By randomising the layout of the process in memory, this location is different for every execution of the application.
2. Background

Even though ASLR can be an effective countermeasure, newer attacks are able to bypass it. A first attack, called heap spraying[41], simply injects a large number of copies of the attacker’s code on the heap, almost filling up the entire process’s address space. To increase the chance of landing on the code, many \texttt{nop} instructions are added in front of the code. This instruction causes the processor to do nothing and move to the next instruction. Thus, a jump to any of these \texttt{nop} instructions results in the attacker’s code being executed. Note that the injection of the code doesn’t require a buffer overflow to occur. The attacker only needs to convince the program to create many objects on the heap, all containing his code. Only a single buffer overflow is required later, to overwrite control data to execute the random jump.

Both ASLR and stack canaries depend on the assumption that the layout and contents of memory are secret. However, a new type of attack, called a buffer overread attack[53], shows that this assumption is incorrect. When a character array that does not end with a 0 byte is printed, the next few bytes (until the first 0) in the application’s memory are printed as well, even though they don’t belong to the printed buffer. If an attacker is able to read the random canary, he can overwrite the return address while remaining undetected, by simply rewriting the same bytes where the canary was. Similarly, for ASLR, an attacker may be able to figure out the location of the buffer by using a buffer overread.

Non-Executable Stack and Heap[55]. Attacks such as a classic stack-based buffer overflow depend on the presence of memory portions that are both writable and executable. A third countermeasure makes the code portion of the program not writable, while making the stack and the heap not executable. As a result, every page in the memory space of the program is either writable or executable, but never both. This security feature is also called $W \oplus X$.

Consequently, attackers have turned to code reuse attacks[39, 57], in which code already present in the program’s memory is combined in such a way that it executes the attack. An example of this type of attack is a return-into-libc attack, where code from the standard C library is used. This library is an interesting target because of its size and because it is present in virtually every program. Since no additional code is injected by the attacker, the protection offered by $W \oplus X$ is rendered useless.

As discussed above, newer attacks are able to bypass many of these countermeasures. Thus, newer countermeasures are being developed constantly, to detect or mitigate these new attacks. However, these countermeasures only protect against a single vulnerability or a small set of vulnerabilities. As N. Frykholm states regarding memory unsafe languages: “No tool can completely solve the buffer overflow problem, but tools can increase the probability that a buffer overflow is detected and reduce the attacker’s chance of successfully exploiting a vulnerability.”[14] Therefore, the attacker model used in this master’s thesis assumes that arbitrary code execution is
2.2 Damage Reduction

The previous section mentioned that buffer overflow attacks are commonly exploited. However, the conclusion of this section was that no countermeasure can efficiently solve the buffer overflow problem. Therefore, a lot of research is being conducted towards reducing the power of an attacker, even if he is able to compromise (part of) an application. This section identifies two main tracks of research.

The first track, described in section 2.2.1, focuses on isolating security-critical portions of applications, often even from the operating system on which they run. This track thus reduces the power an attacker has over the compromised application. Section 2.2.2 discusses the second track. Here the focus is on reducing the privileges of an application or parts thereof, according to the principle of least privilege[44]. This way, the damage an attacker can inflict on the operating system and the rest of its applications is reduced.

The work of this thesis is based on a system from the first track. A mechanism called Self-Protecting Modules is used to isolate security-critical functionality. This mechanism is discussed in chapter 3. In this thesis, however, this system is expanded with a mechanism for privilege containment, reducing the power of each module to the bare minimum to limit its control over the operating system as well. This expansion is discussed in chapter 4. The end result thus belongs to both tracks: an attacker able to compromise a single module will have limited power over both the operating system and the compromised application.

2.2.1 Isolation of Security-Critical Functions

In the first research track, security-critical functions of applications are isolated from the rest of the application. As a result, an attacker that is able to exploit a bug in another part of the application, cannot compromise the security-critical function. These systems thus offer no protection towards the rest of the operating system, but instead focus on protecting small portions of functionality by minimising their Trusted Computing Base (TCB). The TCB of (a part of) an application consists of all the hardware and software components that it needs to trust to ensure correct execution. These systems thus take the legacy operating system, which consists of millions of lines of code[37], out of the TCB of the security-critical function. This kind of isolation can be provided using virtualisation techniques or using hardware modifications.

Systems Based on Virtualisation

A lot of research focuses on using virtualisation techniques to isolate security-critical portions of an application. One of these system is the Nizza architecture[17]. Originally, this system was designed to run entire processes on a separate virtual
2. Background

machine (VM). This way, the legacy operating system is no longer part of the Trusted Computing Base. These applications only need to trust the Virtual Machine Monitor (VMM) and the small kernel upon which they run. However, the Nizza architecture can also be used to securely isolate portions of an application[48]. These portions are extracted into AppCores and executed as a trusted process. The rest of the application continues to run on the legacy operating system. The difficulty of creating such AppCores depends greatly on the interfaces in the application that is to be protected. If the security-critical functionality is clearly separated from the rest of the system, extracting it is relatively easy.

A similar idea is implemented in Fides[51], the unloadable VMM upon which this master’s thesis is based. In Fides, portions of the application, called Self-Protecting Modules, execute in complete isolation of the rest of the application and of the legacy operating system. These modules reside in the same address space as the rest of the process and are protected using a program counter based access control model. The idea of Self-Protecting Modules is described in more detail in chapter 3. The Fides implementation is discussed in more detail in section 3.5.2.

Recent commodity processors gained support for late launch and remote attestation. AMD’s Secure Virtual Machine (SVM) extensions and Intel’s Trusted eXecution Technology (TXT) allow the secure late launch of a VMM or security kernel at any time. On the AMD processor, this can be done using the SKINIT instruction. Whenever this instruction is called, the Platform Configuration Register (PCR) 17 in the Trusted Platform Module (TPM) is reset. Measurements of all software that runs following this instruction are extended in this register, allowing a remote third party to verify that the correct software ran using the value in this register.

The Flicker system[28] uses this hardware support to execute a piece of code, referred to as a Piece of Application Logic (PAL), in complete isolation without the need for a complete VMM. To execute a PAL, the legacy operating system is paused and the SKINIT instruction is called. A measurement of Flicker’s Secure Loader Block (SLB) is then extended into the TPM’s PCR 17, enabling attestation to a remote party that Flicker’s security was enabled while the PAL was executing. The PAL can then run in complete isolation of the rest of the system. A measurement of the PAL is also extended in the PCR 17. When the PAL finishes, a well-known value is extended in the PCR 17, to signal the end of the secure execution. Finally, the legacy operating system is resumed.

Flicker heavily relies on the slow TPM, leading to a significant decrease in performance. This performance cost will probably reduce over time, as the hardware’s performance increases. However, TrustVisor[27], a system based on Flicker, already significantly increases the performance, by simulating the TPM’s functionality on the general purpose CPU. These µTPMs use the hardware TPM as their root of trust, but don’t depend on it as heavily as the original Flicker design.
2.2. Damage Reduction

Systems Based on Hardware Modifications

The same kind of protection can be reached by modifying existing hardware. This approach is used to introduce a program counter based access control model for embedded systems[52]. The access control model is similar to the one used in Fides and thus also to the one applied in this thesis. Because of the performance requirements of embedded systems, the access control model should be implemented in hardware here. New hardware instructions are provided to switch between protected modules. These modules are described in more detail in chapter 3. The hardware implementation is discussed in more detail in section 3.5.1.

The theoretical description of hardware support for SPMs is modified slightly and extended by the Sancus system[31]. More specifically, Sancus adds remote attestation, enabling its usage in systems consisting of multiple nodes in a networked environment, such as wireless sensor networks. Although Sancus is more recent than the hardware implementation described above, it is less similar to Fides and consequently less similar to the SPM mechanism implemented in this master’s thesis. Therefore, in the remainder of this thesis, the implementation described in the previous paragraph is called the hardware implementation of SPMs.

Another system based on hardware modifications is Mondriaan Memory Protection (MMP)[60]. This technique overlays a single address space with several disjoint protection domains. Each protection domain defines access rights to certain parts of memory. These domains enforce control flow by only allowing cross-domain calls to specified points, called switch gates. This system has been used to partition the Linux kernel into protected modules[61].

2.2.2 Privilege Containment

The second research track focuses on reducing the privileges of (parts of) applications. Consequently, an attacker able to compromise a single module or application will not have full control over the rest of the operating system and its resources.

One system that achieves this is Janus[16], a user-level monitor designed to monitor the access of untrusted applications to the operating system. This is done by limiting which system calls the application can execute, disallowing potentially harmful ones. Since Janus only monitors system calls, applications still have full control over their own address space. Because Janus is a user-level monitor, it must run as a separate process to prevent the monitored process from altering its functionality.

A second mechanism designed to reduce the power of an attacker over the operating system is called Protection Wrappers[21]. Here, untrusted applications can be run in a sandbox environment. This sandbox is created by changing the identity of the user that executes the application to that of an unprivileged user. The application will thus have limited privileges over the operating system.
2. Background

2.3 Conclusion

Despite the introduction of numerous countermeasures, crafty attackers continue to find new ways of exploiting applications. As a result of these exploits, attackers are able to execute arbitrary instructions in the application’s memory space. Consequently, the application’s secrets are compromised. Moreover, the attacker can use the operating system’s system calls to affect other applications as well.

Recent research proposes techniques for isolating security-critical portions of applications and for reducing an application’s power over the operating system. In this master’s thesis, these two domains are combined, creating a system that reduces the attacker’s power over both the exploited application and the underlying OS.
Chapter 3

Self-Protecting Modules

Modern applications tend to mix security-critical functionality with other functionality for which security is much less relevant. Web browsers, for example, handle user passwords and cryptographic keys used to protect the secure connections. On the other hand, they also include parsers, portions of software that are likely to include bugs [34, 46, 19, 45].

Chapter 2 showed how attackers can exploit these bugs to gain control over the complete application or even the entire system. Section 2.1 already concluded that the current focus on prevention of these bugs is insufficient, since no countermeasure can efficiently solve the buffer overflow problem. Section 2.2 introduced current research aimed at the containment of these bugs, limiting the damage an attacker can inflict.

In this chapter, one of these new systems, called Self-Protecting Modules [52, 51] is described in more detail. It forms the basis of this master’s thesis. It was originally designed to protect security-critical portions of applications from bugs in the rest of the application or the operating system. In essence, the idea is to introduce a program counter dependent access control model. Thus, access rights to the application’s memory depend on what code is currently being executed. An application and its protected modules share the same address space. Returning to the browser example above, Self-Protecting Modules could be used to protect the cryptographic keys so they can only be read by code designed to use these keys, isolating them from malicious code in the rest of the process.

This chapter starts out with a general description of the concept of Self-Protecting Modules, without specifying any implementation details. Section 3.1 details the layout of an SPM and how it is protected, while section 3.2 focuses on the operations that can be applied to an SPM. Next, in section 3.3, a mechanism for secure communication between protected modules is introduced. Section 3.4 discusses the problem of supporting multiple threads sharing the memory space of processes containing SPMs. Next, section 3.5 describes different ways of implementing hardware or software support for Self-Protecting Modules. Finally, section 3.6 offers the conclusion of this chapter.
3. Self-Protecting Modules

3.1 Module Layout

The basic idea is to isolate security-critical portions of an application from the rest, while they remain within the same address space. This can be done by putting the code and data relevant for the security-critical portion of the application in a Self-Protecting Module. The access rights to memory part of such a module depend on the value of the program counter. Basically, code inside the module has almost full access to its memory, while access by code outside the module is very restricted.

The memory layout of a process containing a single Self-Protecting Module, is shown in Figure 3.1. Blue blocks represent program code, while red blocks indicate data. A darker shade is used for the protected parts of the process. Note that this is a simplified overview of program memory, and doesn’t represent the positioning of these blocks in a real process.

![Figure 3.1: Layout of a process containing an SPM.](image)

A Self-Protecting Module is a part of the process’s virtual memory divided into two sections. The first section is the Public section. It contains information to which all code has read access, but no code has write access. The information is thus integrity protected as soon as the protection is enabled. This section mainly consists of the program’s code. The code is read accessible to the entire process to allow other modules to authenticate this module.

Moreover, full execute permissions over the Public section are only granted for code residing in the module itself. Modules come with a list of predefined entry
3.2. Operations

points. A module can only be called by code outside of the module by jumping to one of these entry points. This protects it against attacks where important checks in the code are skipped.

The second section is called the Secret section. As the name suggests, this section contains all of the module’s secret data. For example, a module designed to encrypt a given input can store its cryptographic key here. Moreover, security-critical control data is stored here as well, including the module’s call stack. Code residing in the module itself has read and write access to the Secret section. However, no access is given to code residing outside of the module. A module can thus completely control which information enters or leaves this section.

While access to the SPM’s data from outside of the module is restricted, the inverse does not hold. Code residing in a module’s Public section has the same access to unprotected memory (all memory not part of any SPM) as code residing in unprotected memory. SPMs are thus free to read from and write to unprotected memory, with the same restrictions that apply to regular code. An overview of the access control rules is given in Table 3.1.

<table>
<thead>
<tr>
<th>from\to</th>
<th>Entry point</th>
<th>Public</th>
<th>Secret</th>
<th>Unprotected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry point</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Public</td>
<td>rx</td>
<td>rx</td>
<td>rw</td>
<td>rwx</td>
</tr>
<tr>
<td>Secret</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unprotected / other SPMs</td>
<td>rx</td>
<td>r</td>
<td></td>
<td>rwx</td>
</tr>
</tbody>
</table>

3.2 Operations

The protection offered by Self-Protecting Modules, as described in the previous section, is provided by a supporting system. The different ways of implementing this supporting system are described in section 3.5. Regardless of the implementation, this system needs to provide some operations for user-level applications to interact with it. Three operations are always required.

Creation When an SPM’s code and data are loaded into memory, they are not instantly protected. Using the creation operation, the supporting system enables this protection. The application must provide a full specification of the layout of the SPM. The supporting system checks that the given layout doesn’t overlap with an existing SPM or the operating system’s kernel.

A unique identifier is generated for the protected module, to allow identification by other modules. The supporting system guarantees that this identifier is not reused until a reboot.

The Secret section is either cleared completely, or its first byte is cleared. This is done to prevent an attacker from injecting false data before the protection
3. **Self-Protecting Modules**

is enabled. By checking the first byte, the module knows that the data in its Secret section cannot be trusted yet. Some modules may then restore data previously stored on disk. Note that this data should be encrypted, but the module cannot know a confidential encryption key yet, since the Secret section is still untrusted. Therefore, the supporting system may provide an additional module with the sole responsibility of securely saving application data. Finally, the memory access control described in the previous section is enabled and the module can start servicing requests.

**Destruction** Once the protection is enabled, it can only be disabled by code running inside the SPM, using this destruction operation. Before destruction, an SPM should clear its Secret section, to prevent any information from leaking when the protection is disabled. Data that must be retained between sessions can be saved securely to the disk.

**Layout** SPMs need to be able to authenticate each other, to pass data securely. The layout operation is given an address and returns the complete layout of the module at that address, enabling other modules to verify that it has been loaded correctly. For example, if its Secret section is smaller than expected (because an attacker tampered with the call to the creation operation), the module may write confidential information in unprotected memory. Thus, a module with an incorrect layout cannot be trusted.

Additionally, a fourth optional operation can be provided to improve the performance of module authentication.

**Testing** This operation uses the unique identifier to verify that a given module is still loaded correctly. The layout operation should thus be used to authenticate a module the first time, while the testing operation is available for subsequent calls.

Finally, a fifth optional operation can be added to simplify full two-way authentication.

**Caller** This operation returns the unique identifier of the caller of this module. If SPM A calls SPM B and SPM B uses this operation, he will receive SPM A’s identifier. This can be used to authenticate the caller and limit access to SPM B’s interface to trusted modules only as described in the next section.

Table 3.2 gives an overview of the three implementations (see section 3.5), which operations they support and the names used for the operations.

### 3.3 Secure Communication

For Self-Protecting Modules to be useful, they should be able to collaborate with other modules securely. Therefore they must be able to authenticate other modules,
to ensure that they can provide them with confidential information and that they can trust the returned result. Moreover, once the authentication is successful, they must have a secure way to pass the information to the other module.

The authentication step is supported by means of a security report, which contains all information about a module that defines its identity. A security report consists of the following three elements:

- A hash of the Public section which allows other modules to verify that the module’s code has not been changed before the protection was enabled.

- The complete layout of the SPM, allowing verification that the creation call was not compromised.

- A cryptographic signature by the creator of the SPM. SPMs can verify this signature using a list of trusted certificate authorities (CAs). Because of this signature, the report can be placed in unprotected memory. Alternatively, it may be placed in the Public section as well (but should be ignored when calculating the hash).

For an SPM to be authenticated, the caller should verify the cryptographic signature, the hash of the Public section and the complete layout of the module, using a call to the layout operation. Subsequent authentications can use the faster testing call if it’s available.

To ensure that confidential information can be passed securely, all parameters should be passed in registers. The only difference with calling a regular function is that the return address cannot be passed on the stack. This is because modules are not allowed to access each other’s stack. Thus, a continuation point address should be passed in a register as well. When this point is called, it should use information stored in the Secret section to ensure that a continuation was expected.

In some cases, a two-way authenticated service call may be required. If the callee passes confidential information in the result, he must make sure that the continuation point is inside a trusted SPM. Therefore, once the callee is entered, he performs a call to the layout operation using the given continuation point. The result is a description of the module layout. Using the security report and the procedure described above, the callee can authenticate the SPM where the continuation point is located.

Table 3.2: Operations in the different supporting systems.

<table>
<thead>
<tr>
<th></th>
<th>Hardware[52]</th>
<th>Fides[51]</th>
<th>Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creation</td>
<td>setProtected</td>
<td>crtSPM</td>
<td>spm_create</td>
</tr>
<tr>
<td>Destruction</td>
<td>resetProtected</td>
<td>killSPM</td>
<td>spm_destroy</td>
</tr>
<tr>
<td>Layout</td>
<td>isProtected</td>
<td>lytSPM</td>
<td>spm_layout</td>
</tr>
<tr>
<td>Testing</td>
<td>/</td>
<td>tstSPM</td>
<td>spm_test</td>
</tr>
<tr>
<td>Caller</td>
<td>/</td>
<td>/</td>
<td>spm_get_id</td>
</tr>
</tbody>
</table>
In other cases, the callee may need to authenticate the caller instead of the SPM where the continuation point is located. This is the case when the callee performs an action where the action itself is security-critical instead of only the returned result. Since the caller can pass any random address as the continuation point, there is no guarantee that it is inside the caller’s SPM. To support this kind of authentication, the supporting system can provide an operation that returns the id of the caller SPM, described above as the caller operation. Alternatively, if the caller operation is not available, the callee can pass a secret token to the given continuation point and refuse service to modules not in possession of this token.

3.4 Multithreading

This section discusses the problems with supporting multiple threads sharing a memory space containing SPMs. This problem is independent of the implementation. Problems specific to each implementation are discussed in section 3.5.

On multi-core systems, two threads sharing an address space can execute concurrently. Even on single-core systems, the operating system scheduler can simulate this concurrent execution. However, this is problematic when Self-Protecting Modules use the secure communication mechanism. In the short time between authentication of the callee and the subsequent jump, the target module may be destructed. Similarly, if two-way authentication is used, the calling module may be destructed right after the callee performed the authentication. Consequently, these modules would issue a jump to unprotected memory and leak secret information stored in the registers. Thus, while one thread runs in an SPM, no other threads sharing this memory space should be allowed to run.

3.5 Implementations

Adding support for Self-Protecting Modules requires a supporting system that provides the protection. There are three possibilities for this implementation. Section 3.5.1 discusses full hardware support[52]. Next, section 3.5.2 describes the Fides system[51], a hypervisor implementation. Finally, section 3.5.3 discusses an operating system kernel implementation, the direction taken in this master’s thesis.

3.5.1 Hardware Implementation

A first way of adding support for Self-Protecting Modules is by modifying the hardware, changing the memory access control model to depend on the value of the program counter as well. Moreover, the operations described in section 3.2, are implemented by adding new instructions to the modified CPU.

There are two big advantages to this type of implementation. The first one is the resulting performance. Since the implementation can be done in hardware, the overhead generated by the modified access control model would be very limited. Since
the program counter dependent access control model is enforced on every memory access, there is no cost of switching between protected and unprotected memory.

The second advantage is the small Trusted Computing Base (TCB). Because support for Self-Protecting Modules is implemented in hardware, only the hardware needs to be trusted. Note that a module can always place explicit trust in a different module by using the secure communication mechanism. As a result, these modules will also become part of the TCB. Nevertheless, the hardware implementation has the smallest TCB of the three implementations.

However, there are also a few disadvantages to an implementation in hardware. First of all, the fact that hardware needs to be modified is a drawback, since commodity systems will not be able to run protected applications. Secondly, an implementation in hardware for today’s high-end CPUs may not be feasible at all, because the access control mechanism would require interaction with the present memory translation mechanisms. Moreover, since the kernel is no longer part of the TCB, support for features such as interrupts, swapping, direct memory access (DMA) or paging would need to be implemented in the SPMs. Porting existing applications thus becomes a lot harder. Moreover the size of a protected application would be considerably larger compared to the original version.

3.5.2 Fides

Another way to implement the Self-Protecting Modules mechanism is by using a hypervisor. The prototype implementation is called Fides and its layout is shown in Figure 3.2. Two virtual machines run on top of the hypervisor, ensuring full isolation between their memory. The two VMs have different access rights to the same physical memory. The first virtual machine, the Legacy VM, contains the legacy operating system kernel. The second one, the Secure VM, runs a small security kernel. Since access rights only need to be switched when transferring between unprotected and protected memory or between two SPMs, the hypervisor can trap illegal memory accesses and switch to the correct VM as needed. The security kernel further implements the memory access control model for the SPMs that it contains.

![Figure 3.2: The Fides prototype. Partially accessible memory regions are shown as hatched regions.](image)

An implementation in a hypervisor has the advantage that it can run on commodity hardware. Moreover, compatibility with existing applications and operating sys-
3. Self-Protecting Modules

tems is retained. Thus, legacy applications and applications that use Self-Protecting Modules can run side by side.

A second advantage of this implementation is the relatively small TCB for the SPMs. Since the modules run on top of a special security kernel that only provides minimal functionality, the legacy operating system is not included in the TCB. Only the hypervisor and the security kernel need to be trusted. Together, they consist of merely 7,159 lines of code, a significant improvement over the millions of lines of code of legacy operating systems.\[37\] Note that the TCB is slightly larger than in the hardware implementation, where only the hardware needs to be trusted. However, this small amount of code can be verified using a formal verification tool, such as VeriFast[20].

The disadvantage of this implementation is its performance. First of all, the virtualisation itself incurs a certain overhead. Thus, even applications that don’t use any modules and thus don’t benefit from the additional security guarantees, will run slower. For Fides, this performance overhead is under 4% for most applications. Secondly, there’s the cost of switching between two VMs, when switching between unprotected memory and an SPM. A macrobenchmark has shown that this performance impact is around 14% for an SSL-enabled webserver. Finally, whenever an SPM is entered, the legacy operating system is paused. This is problematic if operations running inside an SPM run for a long time. In a graphical environment, pausing the legacy operating system can give an unpleasant laggy impression to the user. As a result, Fides cannot be used to execute applications that spend most of their time inside one or more SPMs.

3.5.3 Operating System Kernel Implementation

The third and final way of adding support for Self-Protecting Modules is by modifying the operating system kernel, the approach taken in this master’s thesis.

Similar to the Fides prototype, the modified Linux kernel can run on existing hardware, a significant advantage over the hardware implementation. Existing applications should be able to run on the modified kernel without issues.

A second advantage of an implementation in the Linux kernel is the performance. Compared to Fides, the switching costs should be reduced. Moreover, legacy applications and protected applications run on top of the same kernel, side by side. Since the operating system already implements isolation between different processes, there is no problem with interleaving a process running inside an SPM with other processes. Thus, there is no need to pause other processes when an SPM is entered. This enables new types of protection where the larger part of the application consists of one or more modules.

A third and final advantage is that all existing system calls in the Linux kernel can be used by the protected modules as well. In comparison, the hypervisor implementation doesn’t allow this, since the legacy operating system kernel is not trusted.

The disadvantage of this implementation is that, since the support is implemented in the operating system kernel, the entire kernel needs to be trusted. This results in
a huge TCB. As a result, the kernel implementation offers protection only under the assumption that the kernel cannot be compromised.

3.6 Conclusion

Support for Self-Protecting Modules allows applications to isolate certain portions of their memory space from the rest of the application. Consequently, bugs outside of these isolated portions cannot influence the secrecy of their information. This mechanism is useful, for example, to keep encryption keys safe.

The three ways of implementing support for these modules each offer their own set of advantages and disadvantages. Particularly, an implementation in an operating system kernel will offer fast switching of the access rights. Moreover, new types of applications become possible, since whole applications can now be split up into modules, something that could not be done for applications supported by the Fides prototype. As chapter 4 will show, completely modularised applications open the door for new ways of securing the underlying operating system from intruders.
Chapter 4

Privilege Containment

Aside from an implementation of support for Self-Protecting Modules in the Linux kernel, another important contribution of this thesis is the expansion of this system with a mechanism for reducing module privileges. By reducing a module’s privileges, the damage an attacker can inflict should he compromise the module is limited as well.

However, this privilege containment is only useful if the complete application can be split up in modules. The amount of code running in unprotected memory should thus be limited to initialisation code only. Chapter 3 already indicated that the hypervisor implementation[51] does not support full modularisation, because the entire legacy operating system is paused whenever one process is running inside an SPM. A hardware implementation, on the other hand, would break compatibility with legacy applications. Consequently, in this master’s thesis, the privilege containment mechanism is implemented in an operating system kernel, more specifically in the Linux kernel. The technical aspects of the implementation will be discussed in chapter 5.

This chapter starts by taking a look at the effect an attacker can have on the system if he is able to compromise a single module. This is discussed in section 4.1. Next, section 4.2 discusses how this damage can be reduced further. Finally, section 4.3 concludes this chapter.

4.1 Impact of a Compromised Module

This section describes the damage an attacker can inflict after compromising a single module. Compromising a module in this case means that an attacker exploits a vulnerability in that module. This can only allow an attacker to execute code with the same privileges as this module, or with the same privileges as unprotected memory.

As an example, assume that a process contains an SPM with a buffer overflow vulnerability on its secret stack. Since the module’s Secret section is never executable, the attacker is unable to inject executable shellcode in the module. However, if the module is not too small, it may contain enough code to allow a code reuse attack[39, 57]. In a return-oriented programming attack[40], the attacker combines
small parts of the code, called gadgets, that reside in the SPM’s memory. Each
gadget should end in a return instruction, allowing the attacker to determine the
return address and thus the next gadget to be executed. By combining these gadgets,
code already present in the program is used to construct the shellcode. If enough
gadgets are found, arbitrary instructions and system calls can be executed.

Similarly, an attacker able to compromise a single module is also able to execute
instructions with the privileges of unprotected code. An example attack that achieves
this is an indirect pointer overwrite. In this attack, the attacker is first able to choose
the value of a pointer in the process, for example by overwriting it on the secure
stack after performing a buffer overflow. Later in the execution, a value chosen by
the attacker is written to that pointer. Consequently, the attacker is able to write
any value anywhere in the process’s memory.

To execute arbitrary code in unprotected memory, the attacker can use this
vulnerability to overwrite a return address on the unprotected stack. This is possible
because code inside an SPM has full access to unprotected memory. By pointing
this return address to a second buffer in unprotected memory where the attacker’s
shellcode resides, his instructions are executed as soon as a function in unprotected
memory tries to return to its caller. Note that the attacker has not exploited a
second vulnerability in unprotected memory. He only needs control over the contents
of the second buffer. Thus, by exploiting a vulnerability inside an SPM, the attacker
is able to execute arbitrary code in unprotected memory.

An attacker executing arbitrary instructions in a single module can inflict damage
to the system in two ways. First of all, the attacker’s code can have an effect on the
correct operation of the application to which the compromised module belongs. This
damage is discussed in section 4.1.1. Section 4.1.2 on the other hand, focuses on the
way the attacker can interact with the underlying operating system.

4.1.1 Impact on the Application

Because modules are isolated from each other, an attacker that can execute arbitrary
instructions with the same memory access rights as the compromised module has no
direct access to the Secret section of other modules. Thus, he cannot use his access
to compromise other modules by simply writing different return addresses in their
secure stack.

However, the functionality of modules that explicitly place trust in the compro-
mised module may be impacted. A first example is a module that directly calls the
compromised module. Since the attacker has control over its Secret section, he may
be able to redirect control flow to his own code. This way, he is able to inspect the
passed parameters that may contain confidential information. Moreover, he is also
able to modify the result returned by the compromised module.

Another way for other modules to be affected is if they are called by the com-
promised module. The passed parameters may contain falsified information and the
returned, possibly confidential, information is accessible to the attacker.

Thus, an attacker able to compromise a single module cannot simply expand his
reach to another module’s Secret section. Moreover, because of the authentication
mechanism introduced in section 3.3, he is not able to request services of all other modules. However, he does have the ability to damage the confidentiality and integrity of a limited amount of information stored in or generated by those modules that explicitly trust the compromised module.

4.1.2 Impact on the System

An attacker with control over a single module is able to execute arbitrary instructions as if they run inside the module, or as if they run in unprotected memory. Since both of these domains have the same access to the operating system kernel as a legitimate application, the attacker’s shellcode running here can have a rather large effect on the rest of the system.

The name shellcode originates from its initial purpose[25]. An attacker would inject instructions that perform an `execve` system call, executing the `/bin/sh` program. The operating system would thus launch a shell, possibly with remote access, under complete control of the attacker. The shell is executed as the same user that ran the vulnerable program. Consequently, if the vulnerable program is a daemon running as root, the attacker has full root access to the system.

More generally though, shellcode often uses various other system calls to manipulate the operating system. Using system calls, an attacker can access the file system, execute programs, get input, produce output, exit the current process, ...

Since Self-Protecting Modules in no way limit the access to the operating system, the entire mechanism has no effect on this type of attack. An attacker able to compromise a single module can thus still run any command on the operating system. Because of this threat, section 4.2 discusses a mechanism for reducing the privileges of applications and their modules.

4.2 Reducing Privileges

In section 4.1.2, it was shown that the mechanism of Self-Protecting Modules as described in chapter 3 offers no protection to the rest of the operating system. Consequently, an attacker able to execute code with the privileges of a single module, can still have a considerable impact on the system.

The essential observation here is that shellcode uses existing system calls to inflict damage. Thus, one way to reduce the system-wide impact a compromised application can have, is by limiting which system calls it can execute. I. Goldberg et al. state that “an application can do little harm if its access to the underlying operating system is appropriately restricted”[16].

In this section, privilege containment is discussed. Section 4.2.1 discusses the problems with existing work, where privileges are assigned to complete applications. In section 4.2.2, the addition of a privilege containment mechanism to Self-Protecting Modules is discussed.
4. Privilege Containment

4.2.1 Application Privilege Reduction

The existing work discussed in section 2.2.2 discussed ways to reduce the privileges of complete applications over the operating system. The Janus system\cite{Janus}, for example, cannot control the privileges in a fine-grained manner. As a result, Janus cannot be applied to high privileged applications, despite vulnerability containment being more important there\cite{Vulnerability}.

In fact, the systems discussed in the related work section all limit the privileges of an entire application. This is problematic for applications that require access to many system calls for only a small portion of their functionality. Bugs in other parts of these applications result in the attacker gaining access to all these system calls. Thus, assigning privileges to complete applications is not a sufficient solution.

4.2.2 Modular Privilege Reduction

Since reducing the privileges of complete applications is not sufficient, a mechanism is needed where privileges can be assigned to a subset of an application, in such a way that an attacker able to compromise a single subset cannot execute code under the privileges of a different subset. The mechanism of Self-Protecting Modules fits this description perfectly.

The basic idea is to assign a set of allowable system calls to each SPM. Note that an attacker that has compromised a single module can also execute code in unprotected memory. Therefore the system calls code in unprotected memory can access should be properly restricted as well.

The modules themselves are responsible for limiting their own access rights. Upon creation, each module should notify the supporting system of which system calls it does and doesn’t need. Similarly, unprotected memory should also limit itself as soon as the application is started.

Note that an attacker able to compromise the application before this step, is still able to execute any system call. However, since this is the first thing executed when the application is launched, the only way for an attacker to compromise the application beforehand is by modification of its binary file. This is only possible if the attacker already has access to the system anyway or if the binary file is supplied by the attacker himself through social engineering techniques. Neither of these cases are relevant to this master’s thesis, as the goal is to protect against an attacker that uses an application-level vulnerability to compromise an application and its underlying operating system.

To implement this privilege containment system, the supporting system should add another operation to the list given in section 3.2.

**Disable System Call** The disable system call operation takes a single system call number as parameter. After the call, the system call corresponding to this number can no longer be called from the module that disabled it (or from unprotected memory if that is where this operation is called from). The supporting system should thus keep track of which system calls are still allowed.
and prevent the legacy operating system from fulfilling the request when an attempt is made to execute a disabled system call.

To be useful, applications should be split up in modules completely, leaving only initialisation code or a limited amount of other code in unprotected memory. This immediately indicates why this type of protection cannot be implemented on top of Fides. Because Fides pauses the legacy operating system whenever a single process is running inside a protected module, this type of application would leave the operating system paused for the application’s entire runtime. Therefore, this feature is implemented in the Linux kernel implementation. The disable system call operation is implemented as a system call named `syscall_disable`.

When started, the application will run in unprotected memory. Typically, it will start by enabling the memory isolation of all the modules. Next, the system calls that can be used from unprotected memory are reduced to the minimum necessary. Whenever a module is called for the first time, its initialisation code will run and limit its access to the operating system as well. Section 7.3 describes this type of application in more detail.

After compromising a single module, an attacker is able to inject his own modules and request protection over them as well. Therefore, the access rights of newly created modules should be the same as the access rights of the code that requested the protection. This way, an attacker cannot elevate his privileges by simply creating a new module. Finally, it’s important that once a module has disabled its own access to a system call, there should be no way to re-enable this.

4.3 Conclusion

The mechanism of Self-Protecting Modules as described in chapter 3 does a good job in limiting the power an attacker has over the compromised application and its secrets. However, no effort is done towards reducing the attacker’s power over the rest of the operating system.

This chapter expanded this mechanism with a way for modules to reduce their influence over the operating system. This is done by reducing the system calls that can be executed by each module. Consequently, a compromised module will not allow an attacker to take over the entire system.
Chapter 5

Implementation

The main contribution of this master's thesis is an implementation of support for Self-Protecting Modules with privilege containment in the Linux kernel. The advantages and disadvantages of an implementation in an operating system kernel have been discussed in section 3.5.3.

This chapter focuses on the technical details of the implementation. The mechanism of Self-Protecting Modules was described in chapter 3 and the privilege containment expansion in chapter 4. Chapter 6 will evaluate the implemented solution.

Section 5.1 discusses the Linux kernel background needed to understand the implementation. Section 5.2 describes the data structures used to represent Self-Protecting Modules in the Linux kernel. Next, section 5.3 discusses new functionality. Section 5.4 indicates which existing system calls of the Linux kernel that conflicted with the new security model were changed. Section 5.5 discusses the problem of supporting multithreading in processes using SPMs and describes the solutions that were attempted to add partial support for multithreading. Finally, section 5.6 describes the implementation of the compiler.

5.1 The Linux Kernel

The Linux kernel is an open source operating system kernel, so its source is freely available to anyone. Moreover, the kernel is used on actual production machines, making it a very interesting basis for operating system research. Therefore, the Linux kernel was a logical choice as the basis for an implementation of Self-Protecting Modules in an operating system kernel. This section describes the technical details of the relevant parts of the Linux kernel. All details discussed in this section apply to the Linux 3.6.0-rc5 kernel.

Section 5.1.1 details how the Linux kernel keeps track of the currently running processes. Next, section 5.1.2, describes the structures used to keep track of a process’s virtual memory. Section 5.1.3 briefly introduces how system calls work in the Linux kernel. Finally, in section 5.1.4, the Linux scheduler is explained.
5. Implementation

5.1.1 Processes

The Linux kernel keeps track of all processes currently running on the system[43]. A single process is represented by a task_struct. In fact, each thread has a separate task_struct as well. These structures are chained together in a linked list enabling management tools to fetch a complete list of all processes on the system. Note that this linked list is not used by the scheduler. Processes that should not be scheduled, such as processes waiting for I/O, are still in this list. Section 5.1.4 describes the structures used by the Linux scheduler.

The task_struct contains a lot of process-specific information. Listing 5.1 shows the fields relevant for the rest of this master’s thesis.

```
struct task_struct {
    pid_t pid;
    volatile long state;
    unsigned int personality;
    struct mm_struct *mm;
    [. . .]
}
```

Listing 5.1: Definition of the task_struct.

The first field shown in Listing 5.1 is the pid field. This contains a process identifier that is unique within the kernel.

The state field contains the current state of the process. A value of 0 indicates that it is currently runnable. A negative value denotes an unrunnable process, while a positive value denotes a process that has been stopped.

The personality field contains a process’s execution domain[1]. The execution domain contains a number of per-process options that can be changed by means of a system call. Using this execution domain system, Linux can provide some support for executables compiled using other UNIX-like operating systems. For this thesis, only the READ_IMPLIES_EXEC bit of the personality is relevant. This bit indicates that, for this process, read access to a memory region implies that the memory region is executable as well.

Finally, the mm field references an mm_struct, which contains information about a process’s virtual memory. Section 5.1.2 describes this structure in more detail.

5.1.2 Memory

Like all modern general purpose operating systems, the Linux kernel provides support for virtual memory. Each process thus has its own memory space. All information regarding a process’s virtual memory space is described using two structures: the mm_struct and the vm_area_struct[2].

An mm_struct essentially describes an entire memory space. As mentioned in section 5.1.1, each process references an mm_struct, indicating that that process
runs inside that memory space. Since threads belonging to the same process share their memory space, it’s possible for an mm_struct to be referenced by multiple task structs. Some information about the memory space is included directly in the mm_struct, such as where the text and data segments are located and where the heap and stack start.

However, to describe the entire virtual memory space in more detail, a linked list of vm_area structs is used. The mmap field of the mm_struct points to the first element of the linked list. Each element of the list provides details about a single contiguous memory region. The relations between the different structures are shown in Figure 5.1.

![Figure 5.1: Virtual memory in the Linux kernel.](image)

Apart from the vm_prev and vm_next fields, only the vm_flags field is relevant for this master’s thesis. This field consists of a number of bits, each indicating an option related to the memory region represented by the vm_area_struct. Among these options are the permissions that the process has over the memory region. For each region, a process can be allowed to read, write and/or execute it, depending on the value of this field. Note that since threads share the mm_struct, they also share all the vm_area structs. Consequently, all threads have exactly the same access rights to their shared memory space.

### 5.1.3 System Calls

For the kernel to be able to service user-level applications, some sort of interface to the kernel is needed. Using a software interrupt or a specialised instruction, applications can send a request to the kernel. Each system call has a unique identification number.
The user-level process indicates which service it requests by putting this number in the rax register.

The Linux kernel handles system calls using a table of pointers. Using the system call number as an index in this table, the kernel determines the correct function to jump to. All system calls thus share a single entry point in the kernel, where the dispatching happens.

5.1.4 Scheduling

As in all modern general purpose operating systems, many user-level applications share the processor when running on top of the Linux kernel. To allow for proper multitasking, the operating system has to make two main decisions. First, it needs to decide when the current task is paused, that is, for how long the current task can monopolise the processor. When the task eventually has to give up the CPU, the operating system needs to select which task should run next. These two together determine the scheduling strategy.

The Linux kernel has built-in support for multiple scheduling strategies, called scheduling classes[13]. Each of these scheduling classes provides a single sched_class structure. This structure can be seen as the interface for scheduling classes. It contains a number of function pointers that allow the main scheduler to, for example, request the next task of this scheduling class. Currently, two scheduling classes are provided in the Linux kernel.

The first one is called sched_fair, or the Completely Fair Scheduler (CFS) algorithm[22, 35]. It provides support for the SCHED_NORMAL scheduling policy. This algorithm tries to assign an equal amount of CPU time to each task. Tasks that sleep for a long time, will get their amount of CPU time later and thus receive the same amount as tasks that never sleep.

The other scheduling class that is present in the Linux kernel is the sched_rt class[15]. This is a POSIX compliant real-time scheduler that uses fixed priorities. It provides support for two scheduling policies: SCHED_FIFO and SCHED_RR. The former uses a first-in, first-out queue, where a process will stay on the CPU until it voluntarily releases it. The latter assigns fixed time slices to each process, depending on their real-time priority. A process is allowed to stay on the CPU until the time slice is over.

To pick the next task, the general scheduler iterates over the scheduling classes and asks each of them whether they have a task to run. The first task returned by a scheduling class is selected to run next. The sched_rt scheduling class is checked first, so processes that are assigned the SCHED_FIFO or SCHED_RR policy are always chosen over processes with the SCHED_NORMAL policy.

The tasks themselves are arranged in several run queues, represented by an rq structure. The Linux kernel contains one run queue for each processor in the system. When the scheduling classes are asked to give the next process, they are passed a pointer to one of these rqs, so they know which tasks they have to choose from.
5.2 New Structures

To add support for Self-Protecting Modules in the Linux kernel, a number of new data structures were required. First of all, the kernel must keep some accounting information for each SPM. This information is stored in the new `spm` structure. Listing 5.2 shows the definition of this structure.

```c
struct spm {
    unsigned long secret_start;
    unsigned long secret_end;
    unsigned long public_start;
    unsigned long public_end;

    unsigned long id;

    unsigned long sp;
    unsigned long bp;

    unsigned int entry_count;
    unsigned long *entries;

    char syscalls[68];
};
```

Listing 5.2: Definition of the `spm` struct.

This definition contains five groups of fields:

**SPM Location** The kernel needs to know the location of the Public and the Secret section of the module, to be able to change the access rights when switching between modules.

**Id** Each module has an identifier guaranteed to be unique per process.

**Secure Stack** Contrary to Fides, the modified kernel has support for a secure stack built in. Therefore, the stack and base pointer of a module have to be saved in memory when the module becomes inactive.

**Entry Points** The kernel needs to know what the valid entry points of a module are, to determine whether a jump to a module’s Public section is valid. Therefore, the `entry_count` field contains the number of entry points in the SPM. The addresses of the entry points are stored in an array pointed to by the `entries` field.

**System Calls** The privilege containment mechanism requires the kernel to keep track of which system calls have been disabled for each module. The `syscalls`
5. **Implementation**

array is an array of 544 bits, one for each possible system call number. Note that not all numbers are currently in use.

Apart from the spm structure, the kernel should also keep some information common to all modules within a process. Therefore, a second new structure was introduced: the **spm_task_struct**, shown in Listing 5.3.

```c
struct spm_task_struct {
    unsigned long insecure_stack_pointer;
    unsigned long insecure_base_pointer;

    unsigned long last_id;

    unsigned long prev_spm_id;

    struct element *spm_list;
    struct spm *current_spm;

    char unprotected_syscalls[68];
};
```

**Listing 5.3: Definition of the spm_task_struct.**

This structure keeps track of the following:

**Insecure Stack Location** The stack and base pointer of the stack used by code running in unprotected memory has to be saved when a module is entered.

**Last Id** To guarantee the uniqueness of the identification number, the last used id is saved per process.

**Previous SPM Id** This is used to support two-way authentication. This variable saves the id of the previous SPM that was active in this process.

**SPMs** For each process, the kernel keeps a linked list of all spm structures and a reference to the current spm.

**System Calls** Similarly to the syscalls field in the spm structure, system calls can also be disabled for code running in unprotected memory. This information is kept in the unprotected_syscalls array.

Of course, these new structures have to be integrated with the existing structures in the Linux kernel. Figure 5.2 shows an updated version of Figure 5.1, including the new structures.

The task_struct has been changed to keep a reference to an spm_task_struct. This is a simple one-to-one mapping, so all fields of the spm_task_struct could’ve
been put in the `task_struct` directly. However, to keep SPM-related fields separate, a new structure was used instead. Additionally, the `copy_process` function used by `fork` and its variants was changed to create a new `spm_task_struct` for each new process.

Finally, a new field was added to the `vm_area_struct` as well. This field indicates to which SPM this part of virtual memory belongs. This is important to detect when memory belonging to a protected module is remapped, for example to a file. This problem is described in more detail in section 5.4.2.

### 5.3 New Functionality

The previous section described the new structures that were added to the Linux kernel. Of course, in order to implement support for Self-Protecting Modules, new functionality operating on these structures is required. This section describes all required new functionality, while changes to existing functionality that were necessary to guarantee the security properties are described in section 5.4.

In section 5.3.1, the changes required to enter and exit an SPM are described. Next, in section 5.3.2, the changes introduced to the general system call path are discussed. Finally, section 5.3.3 lists the new system calls that were added and details how they were implemented.
5. Implementation

5.3.1 Entering and Exiting an SPM

Initially, entering an SPM was implemented in the page fault handler. To switch to an SPM, the user-level process can simply jump to one of its entry points. Next, the processor will attempt to execute the instruction located at the entry point. Since the Public section is not executable, a page fault will be generated and the kernel’s page fault handler will be called. This handler was changed to retry the failed instruction if it was located at an SPM’s entry point. Before retrying, however, the access rights are switched so the new SPM becomes active and the Public section becomes executable. Consequently, the instruction will be executed successfully the second time.

However, this method won’t work when exiting an SPM. When returning from an SPM, the destination of the jump can be either a different SPM or unprotected memory. Executing an instruction located in unprotected memory will never generate a page fault. Thus, simply jumping to the return address is not enough to properly switch the access rights.

Therefore, an additional system call was added, called `spm_return`. The only argument of this system call is a destination address. By using this system call, the caller indicates that he wants to jump to a different SPM or to unprotected memory, disabling the current SPM in the process. Note that if the destination is in a different SPM, it still has to be an entry point for the system call to work.

Note that this system call is implemented in such a way that it can also be used to enter SPMs. However, switching to an SPM using a simple jump allows the caller to use more registers for function arguments. When using the system call, `rax` contains the system call number and `rdi` contains the destination address. Moreover, the system call itself will destroy the `rcx` and `r11` registers, according to the x86_64 calling conventions. Because of this advantage of the simple jump, both methods of entering SPMs are still supported in the modified kernel.

5.3.2 System Call Path

The privilege containment mechanism dictates that certain system calls can be disabled completely. As described in section 5.1.3, all system calls in the Linux kernel use a single entry point. To check whether a system call has been disabled, a C function was created that fetches the `syscalls` array of the current SPM (or the `unprotected_syscalls` array of the current process, if called from unprotected memory) and checks whether a given system call number is still valid. The system call entry point was changed to include a call to this function. If the result is negative, the system call is never executed. The entry point immediately returns the `ENOSYS` error code, indicating that no such function is available on the system[18].

5.3.3 System Calls

In section 3.2, the most important operations that can be performed on SPMs have been described. Section 4.2.2 added an additional operation for the privilege containment mechanism. Each of these operations was implemented as a new system
call in the Linux kernel. Additionally, section 5.3.1 introduced another system call
necessary to exit an SPM properly. The following list describes how each of these
system calls was implemented, using the structures described in section 5.2.

**spm_create** A new spm structure is created and added to the list of SPMs of this
process. The access rights to the SPM’s memory are limited, by clearing the
relevant bits (VM_READ, VM_WRITE or VM_EXEC) in the vm_flags field of the
corresponding vm_area_structs.

**spm_destroy** This system call essentially performs the inverse operation of the
spm_create call. The SPM is removed from the list of SPMs in this process
and the original access rights to its memory region are restored. Finally, the
spm structure is freed.

**spm_layout** An SPM is searched based on the given address. If one is found, the
contents of the spm structure of the SPM denoted by the given address are
copied to user memory. Note that the stack and base pointer are not copied,
since these would leak information about the state of the module.

**spm_test** An SPM is searched based on the given id. If one is found, the function
returns success.

**spm_get_id** This system call simply returns the prev_spm_id as saved in the
spm_task_struct of the current process.

**spm_disable_syscall** The bit corresponding to the given system call number
in the syscalls array of the current spm structure is set. If the process is
currently executing in unprotected memory, the unprotected_syscalls array
of the spm_task_struct is used instead.

**spm_return** The current SPM is disabled by changing the vm_flags field of the
corresponding vm_area_structs to represent the limited access rights of an
inactive SPM. If the given address is invalid (e.g. an address in an SPM’s
Public section that is not an entry point), a segmentation fault is injected in
the process. If the address is valid, the kernel has to execute the expected jump.
Therefore, the system call changes the ip register of the user-level process to
the given address before resuming execution in user mode.

### 5.4 Conflicting System Calls

Support for changing memory access rights was already present in the Linux kernel.
Unfortunately, some existing system calls enable modifications of these access rights.
These pose a potential security threat, since they could be used by an attacker to
change the access rights of memory regions belonging to an inactive SPM.

The Linux kernel consists of a huge code base, so manually checking for each
function whether it could harm the security of an SPM is unmanageable. To find
the problematic system calls, the source code was initially searched for code where
the `vm_flags` field of a virtual memory area is changed and for code dealing with protection flags such as `VM_READ` or `VM_WRITE`. In this section, the problems found and their solutions are discussed.

5.4.1 mprotect

The most obviously problematic system call is `mprotect`. Using this system call, a user-level process can request alternative protection over a region of its virtual memory. The requested permissions can either elevate or diminish the access rights of the process. As a result, code running in unprotected memory can simply use the `mprotect` call to enable read access to a module's Secret section.

Luckily, a mechanism to limit what can be achieved using `mprotect` already exists in the Linux kernel. The same `vm_flags` field can be used for this purpose. To disable `mprotect`'s ability to add read, write or execute permissions, the `VM_MAYREAD`, `VM_MAYWRITE` and `VM_MAYEXEC` bits are set and cleared together with the `VM_READ`, `VM_WRITE` and `VM_EXEC` bits.

Additionally, the `VM_MAYSHARE` flag was cleared for memory regions belonging to an SPM, to prevent this memory from being shared between multiple processes. Finally, the `spm_create` system call was changed to check that the memory in which an SPM is created has not been marked as shared beforehand.

5.4.2 mmap

A second problem lies in system calls such as `mmap` and `mremap`. These allow the user-level process to map a portion of its virtual memory to something else. They can, for example, be used to load a file into the process’s virtual memory. The security problem here is that an attacker executing code in unprotected memory could map the Secret section of an SPM to a file under his control. Essentially, these system calls can result in virtual memory areas being deleted or recreated with different permission flags.

To resolve this, the `vm_area_struct` was changed to include a reference to the SPM in which this memory region lies. All allocations of new `vm_area_structs` were changed to set this field to `NULL`. Additionally, whenever the kernel decides to switch to a new SPM, a check is performed to make sure that the SPM’s memory is still mapped correctly. This is done by iterating over the `vm_area_structs` within the SPM’s bounds and making sure their `vm_spm` field is still set correctly. Thus, an SPM that has been compromised by unmapping and remapping part of its virtual memory can never become active. The same check is added to the `spm_test` and `spm_layout` system calls. These now fail when pointed to an incorrectly loaded SPM. As a result, no other SPM will trust a compromised SPM. Note that data residing in the Secret section before it was remapped is still safe, but has become inaccessible to the SPM. Remapping the virtual memory space does not result in the data being copied, only in future reads and writes being redirected to the newly mapped physical memory or file.
Note that an alternative solution would be to actively prevent a module’s memory regions from being unmapped or remapped. However, due to the number and complexity of system calls related to mapping, the implemented solution was deemed simpler and more secure. An attacker can thus only disrupt the service of an SPM using these system calls. However, the techniques introduced in this master’s thesis do not attempt to protect against denial-of-service attacks anyway.

5.4.3 personality

Section 5.1.1 introduced the task_struct’s personality field and, more specifically, its READ_IMPLIES_EXEC bit. The personality system call allows the user-level process to change this field. The READ_IMPLIES_EXEC bit is problematic, since it implies that the entire Public section will be executable from outside of an SPM, thus no longer limiting jumps to an SPM to the valid entry points.

Therefore, as soon as the first SPM is created, this bit is cleared. The personality system call was changed as well, to disallow enabling the READ_IMPLIES_EXEC bit for processes that already contain protected modules.

5.4.4 fork

The final problem is found in the fork family of system calls, including vfork and clone. All these system calls result in the creation of either a new process or a new thread. The newly created task needs to receive a virtual memory mapping. Generally, the kernel has two options on how to assign a new virtual memory mapping.

In the first option, the entire memory space of the parent task (the task that requested the fork) is duplicated to form the new memory space. Thus, all of the vm_area_structs that make up the parent’s memory space are copied. This is problematic for memory regions where an SPM’s Secret section resides. Since information about the presence of the SPM is not copied to the new task, the kernel can’t know that part of the memory space is supposed to be secret.

To resolve this issue, memory regions belonging to Self-Protecting Modules are not copied when creating a new memory space. The Linux kernel has built-in support for this. By adding the VM_DONTCOPY flag to the relevant vm_area_structs, these structures are not duplicated when forking a new task. As a result, the forked task will not have access to the SPM’s memory.

The fork system calls can also decide to share the memory space of the parent task between the parent task and the new task. This happens when the CLONE_VM flag is set. As shown in Figure 5.3, both task_structs will simply refer to the same mm_struct. Since threads share their memory space, creating a new thread will always result in this type of fork. The problem with this kind of memory sharing lies in the fact that one thread may be running in unprotected memory, while a different one has entered an SPM. The result is that the thread running in unprotected memory has full access to the protected module.

This problem was resolved by disallowing the CLONE_VM flag when the parent process already contains an SPM. The system call will return with a permission
error. Additionally, the `spa_create` system call was changed to check for shared `mm_structs` before creating the SPM.

### 5.4.5 madvise

As mentioned in the previous section, the `VM_DONTCOPY` flag is set on virtual memory areas that are part of an SPM, to prevent this memory from being copied to forked processes. As with the other protection flags, this flag was already present in the Linux kernel. Thus, existing code could change this flag as well.

An additional search through the source code was performed, looking for changes to the `VM_DONTCOPY` flag. User-level processes are able to set the value of this flag using the `madvise` system call. To counteract this problem, an additional check was introduced in this system call, to prevent it from clearing the `VM_DONTCOPY` flag on memory regions belonging to an SPM.

### 5.5 Multithreading Issues

As mentioned in section 3.4, the mechanism of Self-Protecting Modules has problems with supporting multiple threads. In the Linux kernel implementation, these problems are aggravated by the fact that the access rights to memory are actually changed. Consequently, if one thread runs in unprotected memory while another is inside an SPM, both threads will be able to access the SPM’s Secret section. Thus, section 5.4.4 introduced additional checks in the `fork` family of system calls to prevent memory sharing in processes containing SPMs.
However, before these checks were introduced, numerous attempts were made to implement partial support for multithreading. The idea is based on the fact that multiple threads running in unprotected memory concurrently do not pose a problem. The solution is to stop all other threads sharing the same memory space as soon as one of them enters an SPM. The hypervisor implementation\cite{51} supports this as well, since the entire legacy operating system is paused as soon as one thread enters an SPM.

This seems like a rather trivial task, but it proved hard to accomplish in the Linux kernel. The complexity of the problem increases significantly on multi-core processors, where inter-processor interrupts (IPI) are needed to stop threads running on a different core. The following list presents an overview of the different solutions that were attempted and their problems:

**Deactivate Task** In a first attempt, the `deactivate_task` function was used to remove the task from its run queue. This proved to be problematic on multi-core processors, as the task could be running on a different core. If the target core was currently accessing the run queue as well, this led to synchronisation issues.

**Use IPI** Building on the first attempt, an inter-processor interrupt was used to make sure the correct CPU called the `deactivate_task` function. This was achieved using `smp_call_function_single`. However, this solution proved to be problematic as well, resulting in cases where the task was not actually stopped or in the operating system locking up. The root cause of these problems was never found.

**Modify Scheduler** The third solution was to modify the `__schedule` function. Whenever a task was selected, the modified scheduler would check whether this task conflicts with any of the other tasks currently executing inside an SPM. If so, the `__schedule` function was restarted from the beginning. This led to problems where the same task was selected over and over again, resulting in an infinite loop in the scheduler and eventually a frozen OS.

**Wait for Scheduler** A final attempt set the task’s state to a paused state and used the `wait_task_inactive` function to simply wait for the scheduler to deactivate them. However, this was problematic for tasks that were already waiting for something else (e.g. the acquisition of a lock in the kernel). These tasks would be activated as soon as their lock was freed, resulting in two threads executing concurrently.

In conclusion, multithreading support has not been added to the Linux kernel implementation yet. The current solution simply prevents new threads from being created in a process containing SPMs. Similarly, the creation of SPMs in a shared memory space is prevented as well.
5. Implementation

5.6 Compiler

To ease the creation of applications using Self-Protecting Modules, a compiler was created that compiles annotated standard C code to SPMs. It was adapted from the compiler created for the Fides prototype [51]. The general structure of the compiler has not been changed. However, due to the technical differences in the implementations, a few details differ.

First of all, the Linux kernel implementation swaps the stack and base pointers while in kernel mode. Under Fides, the compiler takes care of this, inserting the required instructions in the module. Consequently, in the new compiler, the return address can’t be passed in the \texttt{rbp} register as is done in the Fides compiler. The contents of this register are changed by the kernel when switching between SPMs. Thus, in the new compiler the return address is passed in register \texttt{r13}.

Another difference is the way callbacks are handled. In Fides, this is done using a simple jump. This is not allowed in the Secure VM and thus traps to the hypervisor. In the Linux kernel implementation, this jump doesn’t cause a page fault, since unprotected memory always remains executable. Consequently, the SPM is not exited in a secure way. To fix this, the \texttt{spm\_return} system call has to be used instead of the jump. Calling this system call requires specific parameters to be put in the \texttt{rax} and \texttt{rdi} registers. Thus, when executing a callback, the values of these registers are temporarily transferred to \texttt{r12} and \texttt{r13} respectively. As soon as unprotected memory is reached, they are transferred back to the original registers.

The compiler was used to successfully compile a modified version of GNU zip (gzip), introducing a single SPM. This modified gzip was used as a macrobenchmark. The results of this benchmark are discussed in section 6.3.2.
Chapter 6

Prototype Evaluation

An implementation of Self-Protecting Modules in the Linux kernel as described in chapter 5 has several advantages when compared to the existing hypervisor implementation. The first one is an expected improvement in performance. The overhead introduced by the hypervisor makes calling an SPM an expensive operation. More importantly, the hypervisor implementation also affects the performance of legacy applications. Aside from the virtualisation overhead, the hypervisor completely pauses the legacy operating system, affecting all other processes running on it. The second advantage is the possibility for new types of protection, because of the performance increase, but also because legacy applications continue to run as usual when an SPM is entered by a different process.

While the possible applications are described in chapter 7, this chapter focuses on the evaluation of the prototype implementation. Both the performance and the security of the implementation are evaluated.

The performance benchmarks in this chapter were executed on a Dell Latitude E6510 equipped with an Intel Core i5 560M processor running at 2.67 GHz and 4 GiB of RAM. The operating system used is Ubuntu Server 12.04, with both an unmodified and a modified version of the Linux 3.6.0-rc5 x86_64 kernel installed.

Section 6.1 evaluates the performance impact of the changes to the Linux kernel on legacy applications. Next, section 6.2 describes the results of a few microbenchmarks used to obtain more fine grained performance results. In section 6.3, two applications are secured and their performance is evaluated and compared to the original version. Next, section 6.4 evaluates the security of the prototype. Finally, section 6.5 concludes this chapter.

6.1 SPEC Benchmark

An important design goal of the implementation in the Linux kernel was to minimise the performance impact on legacy applications. Ideally, applications that don’t benefit from the additional security guarantees offered by the mechanism of Self-Protecting Modules should not be affected at all. The additional checks introduced in the general system call path (see section 5.3.2) as well as the individual checks in the
conflicting system calls (see section 5.3.3) could have an impact on the performance of legacy applications.

The SPEC benchmark [50] is used to measure this impact on the system’s performance. More specifically, the Integer Component of SPEC CPU2006 (CINT2006) was used. Figure 6.1 shows the relative results of this benchmark. Negative values indicate that the modified kernel was slower. Some benchmarks performed slightly better when running on the modified kernel, while others performed slightly worse. The maximum deviation of the results for the unmodified kernel is a little over 0.4%. None of these differences are truly significant. The SPEC benchmark thus shows that the performance of legacy applications is not affected by the modifications to the Linux kernel.

![Figure 6.1: Relative results of the SPECInt 2006 benchmark.](image)

6.2 Microbenchmarks

To take a closer look at some aspects of the implementation, a few microbenchmarks were developed. Each of these benchmarks focuses on the performance of a single part of the implementation. In section 6.2.1, the effect of the additional instructions in the general system call path is studied in more detail. Section 6.2.2 compares the different ways of entering an SPM. Finally, section 6.2.3 compares calling a regular C function with calling an SPM and performing a system call.

6.2.1 System Call Path

Section 5.3.2 discussed the changes made to the general system call entry point. Though the SPEC benchmark discussed in section 6.1 did not show any significant differences, a microbenchmark was developed to take a closer look at the effects. A
new system call, `sys_nop`, was added to the Linux kernel. As the name suggests, this system call has absolutely no effect. It simply returns success as soon as it’s called. This system call was added to the unmodified kernel as well as to the kernel with support for Self-Protecting Modules.

The results of this benchmark are shown in Table 6.1. The modifications to the system call path introduce an overhead of around 22%. Note, however, that on our test machine, this corresponds to a slowdown of only 8.05\(\mu s\) per system call. Moreover, for real system calls the same absolute overhead is present. Since real system calls will take much longer to execute, the relative overhead is much lower, which explains why the SPEC benchmark didn’t indicate a significant difference in performance.

\begin{table}[h]
\centering
\caption{System call overhead.}
\begin{tabular}{lrr}
\hline
Kernel & Cycles & Relative \\
\hline
Modified & 118488 & 1.22 \\
Original & 96985 & 1 \\
\hline
\end{tabular}
\end{table}

### 6.2.2 Entering an SPM

The second microbenchmark compares the different methods of entering an SPM. The first method uses the modified page fault handler. The caller simply jumps to an entry point, causing a page fault. The handler will notice that the destination address is an entry point and switch the access rights. The second method uses the `spm_return` system call to enter an SPM. The caller supplies this system call with the destination address. Inside the kernel, the access rights are switched and the `ip` register of the user-level process is changed to the given address. Both methods thus use a different way of entering kernel mode.

The results of this benchmark are shown in Table 6.2. Entering an SPM using the `spm_return` system call is faster than simply jumping and causing a page fault. The microbenchmark that follows will thus use this method to enter an SPM.

\begin{table}[h]
\centering
\caption{Entering an SPM.}
\begin{tabular}{lrr}
\hline
Method & Cycles & Relative \\
\hline
Page Fault & 4662480 & 1.16 \\
System Call & 4024227 & 1 \\
\hline
\end{tabular}
\end{table}

### 6.2.3 Comparison Between Function Call, SPM and System Call

To measure the cost of switching the memory access rights, the time it takes to call an SPM is compared with the time it takes to call a regular C function. Moreover, the same result is compared with the time it takes to perform a system call. This
6. Prototype Evaluation

microbenchmark reuses the `sys_nop` system call introduced in section 6.2.1. The function and the SPM that were used simply return as soon as they’re called as well. The SPM does this by immediately calling the `spm_return` system call.

Table 6.3 displays the results of this microbenchmark. A call to an SPM is about 700 times slower, compared to calling a regular function. This overhead is attributed to the overhead of switching to kernel mode and back twice, as well as to the time it takes to switch the memory access rights. Compared to calling a system call, the secure module is only 40 times slower.¹

<table>
<thead>
<tr>
<th>Type</th>
<th>Cycles</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure Module</td>
<td>4024227</td>
<td>677.02</td>
</tr>
<tr>
<td>2 System Calls</td>
<td>96985</td>
<td>16.32</td>
</tr>
<tr>
<td>Function Call</td>
<td>5944</td>
<td>1</td>
</tr>
</tbody>
</table>

Because it’s so expensive to call an SPM, there is an important tradeoff to be made when designing applications that use this protection. On one hand, small SPMs offer strong security guarantees, but introduce a high performance overhead due to the many switches. On the other hand, larger SPMs offer weaker security guarantees, but have a better performance.

6.3 Macrobenchmarks

In this section, the performance overhead in two real applications is evaluated. In section 6.3.1, an SSL-enabled web server is transformed to use a single Self-Protecting Module. Section 6.3.2 protects the extraction of archives using gzip in a similar way.

6.3.1 SSL Web Server

For the SSL-enabled web server, the secret data that needs to be protected consists of the server’s long time secret on one hand and the current connection’s session information on the other hand. Bugs in code running outside of the protected module no longer allow an attacker to hijack a connection or steal the server’s secret.

This benchmark is based on the benchmark used to measure the performance of the Fides prototype[51]. The secure module was created using the PolarSSL cryptographic library[32] and a subset of the diet libc library[58]. Due to the problems with implementing support for multiple threads described in section 5.5, the benchmark was changed to use a separate process to handle each connection instead. Each process has its own protected module. All processes share a portion of unprotected memory with the process that accepts incoming connections and divides the work. The web server uses a 1024-bit RSA encryption key to protect its

¹These test results differ from the ones presented in the paper in Appendix A. The benchmark produced the results presented here with CPU scaling set to performance, instead of on demand. The results presented here are thus more accurate.
6.3. Macrobenchmarks

SSL-connection. Clients that connect to the server are served a simple static 74-byte page.

The Apache benchmarking tool[56] was used to test the performance of both the protected and the unprotected server. Using this tool, a large number of requests can be sent to stress a web server. The concurrency parameter, indicating how many requests should be sent at the same time, was set to be equal to the number of processes used in the web server. For each test, 5000 requests were sent to both servers.

Figure 6.2 shows the number of requests per second the servers were able to handle for an increasing number of processes. For more than 16 processes, the overhead created by the constant context switches overtakes the benefit of having additional processes handle the connections concurrently. The maximum performance overhead reached is 12.72% at 64 processes.²

![Figure 6.2: HTTPS-Server performance.](image)

6.3.2 gzip

In a second macrobenchmark, the gzip application is secured, by isolating its decompression function. Since this function contains a parser, it is likely to contain exploitable vulnerabilities. These vulnerabilities can be isolated by putting the entire application in an SPM, while keeping the parser in unprotected memory.

However, this section is only concerned with the performance of such an implementation. Moreover, the same number of transitions between protected and unprotected memory can be achieved by putting the decompression function in an

²These test results differ from the ones presented in the paper in Appendix A. The benchmark produced the results presented here with CPU scaling set to performance, instead of on demand and with more requests. The results presented here are thus more accurate.
6. Prototype Evaluation

SPM, while keeping the rest of the application in unprotected memory. Thus, in this macrobenchmark, the decompression function is put in a small SPM. The benchmark was created using the compiler discussed in section 5.6.

Note, however, that this type of implementation is much less secure. The decompression SPM is considered very likely to contain vulnerabilities. Moreover, chapter 4 showed that an attacker able to compromise a module is able to execute code in unprotected memory. Thus, an attacker able to compromise the decompression SPM can take control over the entire application and execute all system calls enabled in both the SPM and unprotected memory.

To execute the benchmark, input files with random content were created and compressed. Their uncompressed sizes ranged from 16 KiB to 64 MiB. The time it takes to decompress each of these files was measured 100 times. Figure 6.3 shows the relative overhead compared with the original gzip.

![Figure 6.3: gzip performance.](image)

Section 6.2.3 showed that calling an SPM is an expensive operation, compared to calling a regular function. Consequently, for small input files, the relative overhead is up to 21.9%. For larger input files, however, the overhead drops and finally reaches -0.5% for the 64 MiB input file. Though larger files require more transitions between unprotected and protected memory, the amount of disk I/O that needs to be performed increases as well. Since disk I/O is slow, this component will dominate the performance measurements. The 0.5% performance increase can be attributed to cache effects.

6.4 Security Evaluation

The most important security feature offered by Self-Protecting Modules and their implementation in the Linux kernel is the full isolation of the modules. A module cannot be compromised if it doesn’t contain a vulnerability itself. This means that modules are not affected by bugs in other modules sharing the virtual address space. Recent research[3] has shown that protection mechanisms such as Self-Protecting Modules can provide full source code abstraction. Consequently, an attacker that
has compromised a module can only interact with the other modules by using their valid entry points.

Moreover, the authentication mechanism discussed in section 3.3 enables modules to limit themselves to only accept calls from modules that would call them during correct operation of the application. For example, if an attacker is able to compromise a parser in a web browser, he may try to call a connection’s encryption SPM. This SPM contains the encryption key used for one of the browser’s secure connections. By calling this SPM, the attacker tries to have his falsified information encrypted and sent to the server, making it impossible for the receiving end to distinguish his data from real data. However, during normal operation, the parser would never issue such a request. Consequently, the encryption SPM will refuse to service the attacker’s request.

Additionally, dividing an application in Self-Protecting Modules already prevents some specific types of attacks. A first example is the classic stack based buffer overflow. As an SPM’s Secret section is never executable, they essentially implement a $W \oplus X$ style protection. Consequently, a second buffer in unprotected memory is needed to inject shellcode. Note, however, that this second buffer should not be vulnerable. The only requirement for this second buffer is that the attacker can control the data it contains. Moreover, SPMs also make it significantly harder to find code usable for a code reuse attack because the attacker is limited to accessing SPMs through their entry points.

6.5 Conclusion

The performance of the prototype implementation in the Linux kernel is better than Fides’s performance in three ways. First of all, there is no noticeable overhead for applications that don’t use SPMs. Moreover, the overhead introduced in applications that do use SPMs is slightly less, at 13% compared to 14% for Fides. The most important improvement though is the fact that the legacy operating system is not paused when an application enters an SPM. Chapter 7 will show that this enables new types of applications that could not run on top of Fides[51].
Chapter 7

Applications

Fides, the existing hypervisor implementation of Self-Protecting Modules, causes a significant performance penalty, even for applications not using the protection mechanism. While an application is running inside an SPM, Fides completely pauses the legacy operating system and all of its applications. Thus, under Fides, a call to an SPM should never take too long before it returns.

Chapter 6 showed that the implementation in the Linux kernel allows legacy applications to run alongside protected applications on top of the same kernel. Moreover, the performance of these applications is not affected even if another process is taking advantage of SPMs. As a result, Self-Protecting Modules no longer need to be brief and can run for as long as they want. This creates new possibilities when securing applications.

This chapter describes different types of applications and examples. In section 7.1, applications that only require a small part of their code and data to be protected are discussed. The applications described in section 7.2 require the exact opposite. Here, small parts of the code are likely to contain vulnerabilities, so the entire application should be protected from these parts. Next, section 7.3 discusses applications that can be completely split up in SPMs, leaving only a minimal amount of code to run in unprotected memory. Section 7.4 describes how SPMs can be used to support in-process plugins. Next, section 7.5 describes Trust Assessment Modules, a specific type of module that monitors the application and assesses its security. Finally, section 7.6 summarises all these applications.

7.1 Securing Critical Portions

The first type of protection offered by SPMs is the isolation of one or more relatively small security-critical portions from the potential vulnerabilities in the rest of the application. Each of these portions will then be converted to a single Self-Protecting Module. Note that the rest of the application can either run in unprotected memory or run in its own SPM as well. For the rest of the section, the remaining code is assumed to be in unprotected memory, as displayed in Figure 7.1. This Figure gives a high-level overview of the memory layout of this type of application, ignoring details
about actual locations of code and data. The module contains some sensitive or secret data and a small amount of code to operate on this data. An attacker can only gain access to the secret data through bugs in the small amount of code running inside the module. The absence of bugs in small amounts of code can be verified much more easily using a formal verification tool, such as VeriFast[20].

Figure 7.1: Protecting small security-critical portions.

An example of this type of application is the SSL-enabled web server, discussed in section 6.3.1. The web server’s long term secret and the connection’s session information are the critical data to protect. An attacker is thus no longer able to compromise this secret or the connection as long as the code running inside the module is bug-free.

More generally, any type of application that requires strong protection over a secret key, such as the keys used in cryptographic operations, significantly benefits from the protection offered by Self-Protecting Modules. Applications that set up secure connections, such as web browsers, email clients or web servers, use these keys to protect the connection. If the key is compromised, any information sent over the connection is accessible to the attacker.

Whether this type of application should run on top of Fides or the modified Linux kernel depends on the average time spent in the module. If this time is short, that is, short enough so freezes are not noticeable to the users, Fides is the better option. The reason for this is Fides’s smaller Trusted Computing Base. When running on top of the modified kernel, the entire kernel needs to be trusted in order for the
7.2 Isolating Vulnerable Portions

secrets to be secure. An attacker that is able to inject arbitrary kernel-level code can simply change the protection bits of the Secret section, giving him access to the key. In Fides, the total TCB is only a little over 7,000 lines.

However, if calls to the module are longer, the Linux kernel implementation should be used. In Fides, long module calls block the entire legacy operating system and all of its processes. In a GUI environment, this results in the entire desktop locking up until the module returns, which is undesirable for obvious reasons.

Finally, this type of application doesn’t benefit much from the privilege containment mechanism, where unused system calls can be disabled per module. Since the majority of the code runs outside of the protected module, unprotected memory will require access to a relatively large number of system calls. Moreover, the code outside of the protected module is deemed the more vulnerable part of the application. As a result, an attacker is more likely to inject code running in unprotected memory, giving him access to all the system calls that are not disabled.

7.2 Isolating Vulnerable Portions

The second type of protection is the complete opposite of the previous type. Here, a small portion of the application is deemed more likely to contain vulnerabilities, compared to the rest of the application. In this case, the larger part of the application should be secured from the smaller part. To realise this using Self-Protecting Modules, the entire application will run in a single module, with the exception of the vulnerable part. This small part will run either in unprotected memory or in a second, much smaller, module. Once again, this section uses the first option, as shown in Figure 7.2. Because of this isolation, if an attacker successfully exploits the vulnerable code, the rest of the application remains protected.

An example of this type of application is a web browser. Web browsers contain various parsers, including but not limited to HTML, XML, JavaScript and CSS parsers. A parser is typically considered very likely to contain vulnerabilities, in web browsers[34, 46, 19, 45], but also in other applications[4, 30]. The additional protection offered by isolating the rest of the web browser from the parser is twofold.

First of all, in-memory secrets kept in other parts of the browser’s memory are safe from bugs in the parsers. These secrets can include cryptographic keys used to secure connections, but also more privacy sensitive data, such as the Document Object Model (DOM) tree of a web page displaying the user’s emails.

Secondly, the privilege containment mechanism can be used to severely limit the rights of unprotected memory, where the parser’s code resides. Note that if the parser runs inside a protected module as well, the access rights of this SPM and unprotected memory together make up what an attacker can do from inside of the module. Either way, an attacker that is able to inject arbitrary code is limited in which system calls he can execute. Since typical shellcode contains system calls[59] not needed by a parser, such as the execve system call, the shellcode won’t run properly. Thus, even applications that don’t contain sensitive information themselves can use SPMs to protect the rest of the system.
Both this section and the previous section describe the usage of a single protected module to isolate portions of the application that require additional protection, either because they contain special data or because a small part of the application is particularly vulnerable. It seems the only difference is the size of the protected module. However, this greatly impacts the choice for the underlying system. Small modules, such as those used in the previous section, can run on top of Fides, as long as the execution of such a module remains short.

In this section, however, the entire application, with the exception of the vulnerable portion, is put inside a single SPM. As a result, the application is expected to run inside this SPM for most of the time. Consequently, this type of application should not be executed on top of Fides. Since Fides pauses the entire legacy operating system, other applications would only get a chance to execute while the protected application is running the small amount of code in unprotected memory. This will lead to an unresponsive legacy operating system.

The benefits offered by the privilege containment mechanism are another reason to prefer running this type of application on top of the Linux kernel implementation. Since Fides doesn’t support this, exploiting the vulnerable code would still lead to the execution of arbitrary system calls, potentially giving an attacker remote access to the machine.
7.3 Completely Modularised Applications

A third and final type of protection supported by Self-Protecting Modules is the complete separation of an application in modules. This generalises the two specific cases discussed in the two previous sections, where each application was split up in a single module and unprotected memory (or two modules).

Since an application is always started without any modules loaded, a small amount of code will always need to reside in unprotected memory. Figure 7.3 shows the typical memory layout for this type of protected application. The initialisation code will start out by enabling the protection of all the modules. Next, the \texttt{spm\_disable\_syscall} system call is used to disable all system calls for unprotected memory. This is possible since the application will remain inside the protected modules for the rest of its lifetime. Next, the application enters the first protected module and starts executing.

![Figure 7.3: Securing applications by complete modularisation.](image)

The size and number of the modules depends on the required security guarantees and acceptable performance overhead. A smaller number but larger modules means that fewer transitions can be expected, reducing the performance overhead introduced by switching between the modules. The boundaries of each module should be examined carefully so as to minimise the number of transitions. If function \texttt{foo} is always called by function \texttt{bar}, the two should probably reside in the same module, as long as neither of them either requires additional protection or is at high risk of
The downside of having larger modules lies in the usage of the privilege containment mechanism. Modules containing a relatively large amount of code will generally require continued access to more system calls. Moreover, the number of bugs in code has been found to be proportional to the size of the code\cite{29}. Therefore, larger modules not only increase the damage an attacker can inflict if they become compromised, they are also more likely to become compromised.

This type of protection is more intrusive and harder to implement than the ones discussed in the previous sections. As mentioned above, dividing the code between modules can be tricky. Additionally, great care must be taken to properly assign the data to the modules where they belong, where the code operating on the data is located. Data that must be shared among modules should either be put in unprotected memory, or communicated between the modules constantly. Applications that have been developed with modularity in mind will obviously make better candidates for this kind of protection.

The examples given in the previous sections can also benefit from the additional security guarantees offered by this type of protection. Web browsers, for example, were mentioned in section 7.1, since they contain cryptographic keys used to protected secure connections. They were also discussed in section 7.2, focussing on the possibility of vulnerabilities in their parsers. Combining these two ideas already results in the browser being split up in multiple modules. The remaining code and data can either be put in a single module, or split up in different modules as well. This depends on the tradeoff between security and performance, as mentioned above.

Similarly to section 7.2, this type of protection cannot be implemented when running on top of Fides. As the application spends no time in unprotected memory, the legacy operating system won’t get a chance to continue. Moreover, when running on top of the Linux kernel implementation, these applications can make extensive use of the privilege containment mechanism to limit the damage an attacker can inflict, should he compromise one of the application’s modules.

### 7.4 In-Process Plugins

The mechanism of Self-Protecting Modules can also be used to support in-process plugins. Because modules inherit the access rights of the module that creates them, the access rights of these plugins can be limited by the application itself. Figure 7.4 shows the memory layout for supporting in-process plugins.

The displayed layout consists of three modules. The first module is the application itself. Note that there is no reason why this should be a single module. The application itself can be split up in more modules, using either of the three types of protection discussed above. The second module is the plugin loader. This module only consists of code and has the responsibility of enabling the protection of a plugin. The access rights of this module are limited to the access rights that the plugins should receive once they are loaded. Consequently, an in-process plugin cannot execute arbitrary system calls. The third and final module is a plugin. In a real application, more
than one plugin is probably present in memory.

Because the entire application runs inside different SPMs, the performance drawbacks of the hypervisor implementation outweigh its security advantages. Moreover, the privilege containment mechanism is needed to limit the plugins' impact on the operating system. Therefore this type of application is only supported by the Linux kernel implementation.

### 7.5 Trust Assessment Module

This section discusses another option, where the entire application runs in unprotected memory as usual. Additional code is added to the process inside an SPM, running alongside the application. The goal of this module is to assess the current security status of the application.

This kind of module is called a Trust Assessment Module (TAM), referring to the goal of indicating the feasibility that the application can still be trusted by either the user or a communicating application. Obviously, the module needs to be isolated from the rest of the application. Otherwise, an attacker that has compromised the application would be able to change memory belonging to the TAM as well. Moreover, the module requires access to the application's memory. The isolation offered by Self-Protecting Modules has these characteristics, since modules run in the same virtual memory space as the rest of the application. Figure 7.5 gives an
overview of the memory layout of a process with a Trust Assessment Module.

![Figure 7.5: A Trust Assessment Module protected using Self-Protecting Modules.](image)

Fides is a slightly more suitable protection layer for Trust Assessment Modules than the Linux kernel implementation. Even though Fides affects the performance of legacy applications, the following three reasons support the decision:

**Additional Security** Fides offers additional security guarantees, isolating the TAM from the legacy operating system’s kernel as well. To trust the TAM, its isolation needs to be guaranteed, so the small TCB for modules running on top of Fides is a significant advantage. Moreover, when running under Fides, the trust of the legacy operating system’s kernel can be assessed by the module as well. When running on top of the modified Linux kernel, this would not make sense, since the module would assess the security of memory that it needs to trust to be able to perform its task properly.

**Performance** The assessment module should try to have a minimal impact on the performance of the assessed application. Consequently, assessing the security should not take long. Whenever the TAM is called, it should return within a relatively short time. Therefore, the fact that Fides pauses the legacy operating system when a module is entered is not a problem.

**Privilege Containment** This system doesn’t benefit from the privilege containment mechanism present in the Linux kernel implementation. Since the TAM needs to be trusted, there is no need to reduce its privileges.
Apart from deciding the underlying system, a number of other choices need to be made about the implementation of Trust Assessment Modules. Each of the following sections discusses one of these choices. Section 7.5.1 discusses the options for adding the TAM to the process’s memory. In section 7.5.2, the options for when control should be passed to the module are described. Finally, section 7.5.3 takes a look at what the module could look at to deduct an application’s security status.

### 7.5.1 Adding the TAM

A first question that arises is how the module is added to the process. Several options are imaginable.

In the first option, the programmer creates the module himself and adds it to the compiled application. The application is then responsible for enabling the protection at startup. This is very similar to the type of application discussed in section 7.1, where part of the application requires additional protection. The advantage of this option is that the developer has additional knowledge about his application. He can include extra checks that a more generic module couldn’t perform. For example, the developer may know that a certain linked list never contains more than 10 items. If this is crucial to the security of the application, the TAM can check that this is still correct every time it’s called. The drawback of this technique is that the application binary needs to be trusted at startup. If the executable is modified by an attacker, he may modify the Trust Assessment Module’s functionality. Since the TAM must assess the security of the application, it should not be affected by such modifications.

The other option is a Trust Assessment Module that is injected by the supporting system, in this case Fides. Whenever an application is launched, a small portion of memory is reserved for the module. Fides is then responsible for loading the module and activating the protection. Since the module is part of Fides, it is more generic and will not be able to benefit from application-specific knowledge.

Finally, there’s also the possibility of combining the two options discussed above. The application then includes two Trust Assessment Modules that need to cooperate to detect whether the application’s memory has been compromised. The general TAM injected by Fides performs more generic checks, including a check whether the application’s TAM is loaded correctly. The application’s own TAM on the other hand uses application-specific knowledge to check the application’s memory contents.

### 7.5.2 Calling the TAM

A second crucial question is when the Trust Assessment Module should get control. The most obvious option here is to use a fixed interval. A hardware timer can be used to redirect control to the supporting system, which in turn calls the TAM. A second option is to only pass control to the TAM when a security-critical operation is to be performed. For example, whenever the application issues a system call, the trust of the application could be assessed by the TAM before the supporting system allows the system call to be executed. A combination of these options can be used
as well. Additional research is needed to examine the advantages and disadvantages of these possibilities and of other options not discussed here.

### 7.5.3 Actions of the TAM

Finally, the Trust Assessment Module needs some way to decide whether an application is still trustworthy or not. If the assessed application doesn’t use any additional protected modules, the TAM has complete access to its memory. As discussed in section 7.5.1, modules included within the application are capable of using application specific data. On the other hand, the supporting system can also inject a more generic module.

Both of these modules can use a technique based on the specification of memory invariants, currently used in the detection of kernel-level rootkits[38, 6]. For TAMs created by the application developers, the invariants can be created manually. A single common module injected by Fides, however, cannot directly use application-specific knowledge. Therefore, the invariants should either be general enough to hold for all applications (e.g. invariants related to correct usage of the stack, or invariants related to the kernel) or the invariants should be inferred automatically (e.g. using a system such as Daikon [12]). Future work is needed to look into other possible methods to assess an application’s trustworthiness.

When the Trust Assessment Module detects that an application has been compromised, several possible actions can be taken. Depending on the accuracy of the detection method, the module can immediately signal Fides to terminate the monitored application. However, if there is a risk of false positives, it’s better to simply warn the user. Note that to warn the user, some form of secure I/O is needed, since the operating system’s graphic drivers cannot be trusted by modules running on top of Fides.

### 7.6 Summary

Each of the previous sections presented a different type of application and discussed which supporting system should be used for these applications. Table 7.1 summarises them, indicating how much code is left in unprotected memory, whether the protected application uses the privilege containment mechanism and what supporting system is preferred.

<table>
<thead>
<tr>
<th>Application</th>
<th>Unprot. Memory</th>
<th>Priv. Reduction</th>
<th>Pref. System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical Portion</td>
<td>Large</td>
<td>No</td>
<td>Fides</td>
</tr>
<tr>
<td>Vulnerable Portion</td>
<td>Small</td>
<td>Yes</td>
<td>Linux Kernel</td>
</tr>
<tr>
<td>Full Application</td>
<td>Small</td>
<td>Yes</td>
<td>Linux Kernel</td>
</tr>
<tr>
<td>In-Process Plugins</td>
<td>Small</td>
<td>Yes</td>
<td>Linux Kernel</td>
</tr>
<tr>
<td>TAM</td>
<td>Large</td>
<td>No</td>
<td>Fides</td>
</tr>
</tbody>
</table>
Chapter 8

Conclusion

This master’s thesis started by stating that the number of computers infected with malware worldwide is estimated to be around 30%\[^{36, 5}\]. Malware often spreads by exploiting application-level vulnerabilities, resulting in execution of arbitrary instructions and system calls. Consequently, the prototype developed in this thesis is designed to limit the attacker’s power over the exploited application and the underlying operating system.

This chapter is organised as follows. Section 8.1 discusses the challenges encountered during the implementation. Next, section 8.2 discusses the potential for future work. Finally, this master’s thesis is concluded with a summary of the contributions in section 8.3.

8.1 Challenges

The first challenge encountered in this master’s thesis was to become familiar with the Linux kernel’s huge codebase. Luckily, the kernel’s code is relatively well documented. Moreover, a large amount of online discussions and documentation is available as well and the implementation only required integration with a small part of the kernel. The combination of these factors made this challenge manageable.

The biggest challenge proved to be the implementation of support for multiple threads in processes that use SPMs. This problem was encountered when trying to protect the SSL-enabled webserver described in section 6.3.1. The attempted solutions have been described more extensively in section 3.4. Since none of these solutions proved successful, multiple threads are currently not supported.

The third and final challenge of this master’s thesis was the evaluation of the prototype. To evaluate its performance, numerous microbenchmarks and macrobenchmarks were created, as described in chapter 6. The main challenge lied not in the creation of the benchmarks, but in their execution. To ensure a proper performance evaluation, the factors that influence the performance need to be limited. For the macrobenchmarks, for example, the protected and unprotected applications need to be as similar as possible. Moreover, for all performance benchmarks, the CPU should run at a constant speed. By default, CPU throttling changes the CPU frequency
8. Conclusion

depending on the load, leading to incorrect and varying results. The results presented in this thesis all have the CPU set to run at a constant speed.

8.2 Future Work

This section takes a look at the possible extensions and improvements that can be made on the achieved results.

Performance The first possible improvement consists of optimising the prototype for performance. The current implementation is a prototype developed by a single person with no prior knowledge of the Linux kernel, so there is certainly room for improvement in performance.

For example, the current implementation handles the `mmap` family of system calls by checking whether an SPM’s virtual memory space is still loaded correctly before switching to it. Instead, the execution of these system calls on memory regions belonging to an SPM should be prevented. Consequently, there is no need to check whether the SPM’s memory is still mapped correctly on every switch. This can lead to a significant performance increase when switching between SPMs. A microbenchmark with an SPM that returns immediately showed a performance improvement of about 3% when removing these checks. The results of this microbenchmark are shown in Table 8.1.

Table 8.1: Overhead of memory mapping checks on SPM switches.

<table>
<thead>
<tr>
<th>mmap checks</th>
<th>Cycles</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disabled</td>
<td>3896767</td>
<td>0.97</td>
</tr>
<tr>
<td>Enabled</td>
<td>4024227</td>
<td>1</td>
</tr>
</tbody>
</table>

Applications Based on the implementation in the Linux kernel, new types of protection become possible, as discussed in chapter 7. A second line of future work can create these new applications and evaluate their performance. Moreover, the Trust Assessment Module discussed in section 7.5 needs further research to fill in the details.

Multithreading Support Thirdly, the implementation in the Linux kernel can be improved significantly by adding support for multiple threads. The current prototype disallows forking with memory sharing enabled. By properly stopping all threads belonging to the same process when one thread enters an SPM, this restriction can be lifted.

However, for the new types of applications, where the entire application runs inside one or more SPMs, this does not resolve the problem. For these applications, multiple threads remain problematic, as all threads have the same access rights when only a single thread is running inside an SPM. A possible solution here is to always switch the access rights when the scheduler switches
threads. However, this will introduce a considerable overhead on thread switch-
ing. Moreover, this solution is not valid for multicore processors, since there
two threads can run simultaneously.

**Fine Grained Privileges** A final line of future work can introduce more fine
grained privileges for SPMs. Currently, SPMs can only be limited by disabling
specific system calls completely. Building on this idea, future work can, for
example, only allow specific system calls if specific parameters are passed. This
would reduce the power of an attacker with the ability to execute arbitrary
instructions even further. Ultimately, the attacker can only execute the same
system calls with the same parameters that the module can execute during its
correct execution.

An example of this expansion is to limit modules so they can only execute the
`execve` system call when a specific program is passed. Consequently, widely
used shellcode that uses `execve` to execute `/bin/sh` will not run, even if the
compromised module requires access to that system call to execute a different
application.

### 8.3 Contributions

The focus of this master’s thesis was on the implementation of support for Self-
Protecting Modules in the Linux kernel. Furthermore, the idea was expanded to
limit the attacker’s privileges with respect to the operating system. More specifically,
the following contributions have been made:

- Chapter 4 expanded the idea of Self-Protecting Modules to limit the modules’
  privileges over the operating system according to the principle of least privilege.
  Consequently, the privileges of an attacker able to compromise a single module
  are reduced as well.

- In chapter 5, the implementation of this combined mechanism in the Linux ker-
  nel was presented. This was necessary because of the limitations of the existing
  hardware and hypervisor implementations with respect to full modularisation
  of applications.

- To ease the creation of SPMs that run on top of the Linux kernel, a compiler
  created for the Fides prototype was adapted. This compiler was described
  in section 5.6. Consequently, annotated standard C code can be compiled to
  generate SPMs that run either on top of the Linux kernel or on top of Fides.

- In chapter 6, the implementation was evaluated, showing that the correct
  execution and the performance of legacy applications is not affected by the
  changes introduced in the Linux kernel. Moreover, a macrobenchmark showed
  that the performance overhead for applications that do use Self-Protecting
  Modules is up to 13%.
• Finally, chapter 7 discussed several types of applications that can be supported by Self-Protecting Modules. It was shown that the implementation in the Linux kernel enables new types of protection. Vulnerable portions of applications can be isolated or the entire application can be split up in SPMs. Moreover, the kernel implementation can be used to sandbox in-process plugins.

The original goal of this thesis, an implementation and evaluation of Self-Protecting Modules in the Linux kernel, has been reached. Additionally, the mechanism was expanded to support privilege containment with respect to the operating system and the compiler of the Fides prototype was adapted to support the Linux kernel implementation.
Appendices
Appendix A

Paper
Efficient Application Modularization in the Linux Kernel

Niels Avonds, Raoul Strackx, Pieter Agten, and Frank Piessens

Katholieke Universiteit Leuven,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium,
niels.avonds@student.kuleuven.be
raoul.strackx@cs.kuleuven.be
pieter.agten@cs.kuleuven.be
frank.piessens@cs.kuleuven.be

Abstract. Developing large modern applications in a secure way can be challenging. Modern high-level programming languages include modularization features to cope with the complexity of a large code base, but these features offer no security guarantees as they are lost at compile-time. As a result, a bug anywhere in an application results in the entire address space being compromised.

To address this issue, we propose a new technique for modularization consisting of (1) compile-time changes to create modular software and (2) a modified Linux kernel that enforces isolation between these secure software modules. Moreover, modules are able to reduce their own privileges, limiting an attacker even if a module becomes compromised.

For applications that make use of the modularization, the performance cost is up to 11%, while other applications running on top of the same kernel are not affected.

1 Introduction

Software development is an ongoing process where the needs of the users constantly change. As a result, software products continuously evolve and grow, resulting in an expanding code base. As the number of bugs in code has been found to be proportional to its size [1], this software growth has a negative impact on software quality and software security. At the source code level, a large code base can be managed using modularization features built in to modern high-level programming languages.

However, these features are lost at compile-time since all the assembly code runs in the same memory space. Consequently, the security critical portions of the application’s code execute within the same memory space as the rest of the code. As a result, a bug anywhere in the application may compromise the secrecy of a key or the flow of a security critical operation.

To make matters worse, more and more applications include some mechanism for plugins, allowing third-party developers to add functionality. As a result, code from multiple stakeholders is executed within the same address space. A
plugin containing malicious code or containing a bug that allows the injection of malicious code can compromise the security of the entire application and all other plugins.

We propose a new technique for software modularization that allows developers to securely isolate portions, called modules, of their applications, where modules are able to limit their own privileges. This limits the damage an attacker can cause when compromising a single module. Accessing a module is only possible using pre-defined entry points. The software developer defines the modules and compile-time changes take care of module initialization. At runtime, a modified Linux kernel enforces module isolation by changing access rights to the relevant portions of user-level memory.

More precisely, we make the following contributions:

- We describe a program-counter based access control model that allows for modularity in applications that is retained at compile-time;
- We add the ability to disable specific system calls for specific modules according to the principle of least-privilege;
- We report on an implementation of support for these modules in the Linux kernel, demonstrating that legacy applications and applications depending on our access control model can run alongside each other on top of the same kernel;
- We show that the performance overhead of our implementation is 11% on average for applications making use of the access control model. Furthermore, we show that the performance and correct execution of legacy applications is not affected by our changes to the Linux kernel.

The remainder of this paper is structured as follows. In Section 2 we define our objectives by describing the attacker model and security properties. In Section 3 we introduce secure modules and our program-counter based access control model. Section 4 discusses our implementation in the Linux kernel and in Section 5 its performance is evaluated. Finally, we describe related work in more detail and offer our conclusion.

2 Objectives

Modern operating systems run an entire process in one address space. As a result, every instruction executed in this process, has access to all of the process’ memory. This situation can be harmful when part of the process’ code cannot be trusted, as is the case after the successful exploitation of a buffer overflow vulnerability or in an application with a plugin system. An attacker with the ability to execute arbitrary instructions can potentially expose secret keys or change the code to be executed for some security critical function. We will describe changes to the Linux kernel that introduce a new system for application modularization, significantly reducing the power of an attacker or a plugin by placing security sensitive code and data in separate protected modules.
2.1 Attacker Model
We consider an attacker that can execute arbitrary code in the process’ address space. This kind of access can be achieved in practice by means of an in-process plugin or by exploiting bugs in the application, such as buffer overflows [2]. Using these bugs, the attacker can inject arbitrary code inside the process’ memory [3]. As a result, an attacker can also execute arbitrary system calls, including system calls used to change memory protection. We assume that the kernel is secure, so an attacker doesn’t have the ability to execute arbitrary code in the kernel.

More importantly, an attacker can also enable protection for his own modules. There is thus no guarantee that modules asking for protection can be trusted. As a result, we have to protect modules from one stakeholder from the modules of an other, possibly malicious stakeholder as well, instead of only from code executing in unprotected memory.

We assume an attacker that has remote access only and is thus not able to perform a physical attack on the hardware.

Finally, regarding the cryptographic primitives that are used, we assume the standard Dolev-Yao model. [4] An attacker can observe, intercept and adapt any message. Moreover, the attacker has the capability of creating messages, for example by duplicating observed data. However, the cryptographic primitives used cannot be broken.

2.2 Security Properties
The adapted kernel gives application developers the ability to create protected modules. It enforces the following security properties:

Restriction of entry points It should be impossible to jump to an arbitrary position in a module from the outside. A module should only be called from unprotected memory or by another module by jumping to one of the legal entry points in this module.

Security and confidentiality of module data A module’s secret data, such as cryptographic keys, should only be read and modified by code executing inside the module itself.

Authentication of modules Modules should be able to identify other modules within the same memory space in a secure way.

Secure communication between modules Once authenticated, two modules should be able to pass messages to each other in a secure way, preserving integrity and confidentiality of the messages.

Privilege Containment Each module is responsible for disabling all system calls that it no longer needs, limiting the privileges of an attacker that can execute arbitrary code in that module.

Note that it is not our goal to protect modules from vulnerabilities within the module itself. In other words, modules containing exploitable bugs, including logical faults (e.g. a faulty API design [5]) and implementation-level vulnerabilities [3], will remain exploitable with protection enabled. However, we do want
to protect modules from similar vulnerabilities in other modules that it does not explicitly trust or in unprotected code sharing the same address space.

3 Protected Modules

All modules in an application are executed within the same virtual memory space, alongside code in unprotected memory. To protect the modules from any external code, we use a memory access control model where the access rights to memory depend on the value of the program counter. Roughly speaking, the memory of a protected module is only fully accessible from within that module. The value of the program counter is used to determine in which module the processor is executing. Code residing in memory outside of this module only has limited access to the module’s memory.

In this section, we discuss how modules are isolated from each other. In Section 3.1, we describe the layout of a module and detail what protection our kernel offers to the modules. Next, in Section 3.2, we describe the system calls used to perform operations on the modules. In Section 3.3, we discuss the lifecycle of a module. Section 3.4 describes the privilege containment mechanism. Finally, in Section 3.5, we discuss how the modules can interact in a secure way.

3.1 Module Layout

![Diagram of a protected module layout](image)

**Fig. 1.** The layout of a protected module.

The basic layout of a module, shown in Figure 1, is an area of the virtual memory divided in two sections. To allow for proper protection, each of these sections must be page-aligned. The *Secret* section contains a module’s private data. This data contains secrets important to the module, such as cryptographic keys, as well as data relevant to the correct execution of the module, including
A module also comes with a list of memory addresses that are valid entry points in its Public section. The modified kernel enforces that a module can only be activated by jumping to one of these entry points. The complete set of rules enforced on the process' memory is shown in Table 1.

### 3.2 System Calls

To support protected modules in our modified kernel, some additional operations need to be supported to create, destroy, authenticate and exit a module. Each of these operations is implemented as a new system call. We also added the ability for modules to disable certain system calls when they are no longer needed.

**mod_create** This system call is used to create a new protected module. The caller supplies the location and size for the Public and Secret sections. Before initializing the module, the operation checks whether the supplied parameters are valid. The memory locations where the module will be created, should not be mapped to a file or shared between processes. Furthermore, modules may not overlap each other or the kernel. If all these checks pass, an id is generated for the module, allowing other modules to authenticate it. This id is guaranteed to be unique by the kernel until a reboot. Next, the first byte of the module’s Secret section is cleared. This way, modules know that their secret section has not yet been initialized and that they cannot trust its contents yet. Finally, the kernel activates the memory protection of the new module and returns control to the application.

**mod_destroy** This system call allows a protected module to destruct itself. The operation takes no parameters and will thus determine which module

---

<table>
<thead>
<tr>
<th>from/to</th>
<th>Entry pnt. Public Secret unprot.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry pnt.</td>
<td>x</td>
</tr>
<tr>
<td>Public</td>
<td>rx</td>
</tr>
<tr>
<td>Secret</td>
<td></td>
</tr>
<tr>
<td>Unprot./</td>
<td></td>
</tr>
<tr>
<td>other module</td>
<td>rx</td>
</tr>
</tbody>
</table>
to destroy by looking at the program counter. If called from unprotected memory, this call will do nothing and return an error condition. It’s important that a module cannot be destroyed from the outside, since this would allow an attacker to tear down the protection and read the contents of the Secret section.

**mod_layout** This system call is used to support module authentication. Given any address inside the process’s virtual address space, this operation will return the complete layout and the id of the module at this location, but only if the module is still loaded correctly. This means that the kernel will check whether no pages of the module have been unmapped or remapped.

**mod_test** This system call is similar to **mod_layout** but is more efficient. Given an id, it indicates whether the module is still loaded correctly.

**mod_return** This system call is required to exit a module. To restore the access rights, control needs to be transferred to the kernel. This is done using the **mod_return** system call.

**syscall_disable** This system call is passed the number of the system call that should no longer be allowed for this module. Any attempt to call that system call afterwards will result in a permission error. There is no way for a module to reenable the disabled system calls. We elaborate on this in Section 3.4.

### 3.3 Life Cycle of a Module

![Diagram](image.png)

**Fig. 2.** The life cycle of a protected module.
In this section, we describe the processes of initializing and destroying modules. Both processes are shown in Figure 2.

Module initialization. The process of setting up a module is shown in Figure 2a. In the first step, an application loads the module’s code in insecure memory. Each section needs to be placed in a contiguous chunk of virtual memory, and the start and end of the section have to be page aligned.

In step 2, the application requests the protection over the module, using the `mod_create` operation. The operating system takes the required steps as described in Section 3.2. Note that an attacker that compromised the module before activation of the protection will be detected and mitigated later on. The module is now protected and can be authenticated and used by other modules.

The first call to a module will include the module initializing itself. The `mod_create` call set the first byte of the secret section to zero, indicating to the module that it hasn’t initialized itself yet. Initialization consists of three steps. First, the module changes the first byte of its secret section, indicating that it no longer needs to perform the initialization next time it’s called. Next, any module-specific initialization actions are performed. For example, the module may need to retrieve a previously stored secret. Finally, the module disables all the system calls that it won’t need using the `syscall_disable` call, to avoid abuse of these system calls by an attacker.

Module destruction. The destruction of a module, shown in Figure 2b, can only be initiated by the module itself. First, a protected module overwrites its Secret section, to avoid leaking any confidential information. Then, the module uses the `mod_destroy` operation to disable the memory protection.

3.4 Privilege Containment

An important feature of our modules is that each module is limited in which system calls it can execute according to the principle of least privilege. An attacker that is able to compromise a single module thus will not be able to execute arbitrary system calls. Rather, the attacker is limited in the same way the module is.

When a module is called for the first time, it will notice that it has not been initialized yet, by checking the first byte of the secret section. The module will then perform some actions specific to its needs, after which it disables all the system calls that it no longer requires. The `syscall_disable` system call does exactly this. Note that once a system call has been disabled for a module, there is no way for the module itself, or for an attacker that compromised the module to enable it again.

The privileges of unprotected memory can be limited in exactly the same way. Any application will start by running code that resides in unprotected memory. A typical protected application will first set up all the modules and then jump to an entry point. Just before this jump, all system calls can be disabled for
unprotected memory, since the application will never return to it. This way, buffers residing in unprotected memory can no longer be used to inject code containing system calls.

Newly created modules will be limited in the same way as the code that created it. An attacker injecting his own module will thus not gain any additional privileges.

3.5 Secure Communication between Protected Modules

The possibility of secure collaboration between the modules of an application is important. To achieve this, modules must be able to authenticate each other and establish secure communication channels. This section describes both actions in more detail.

To make it easy for a third party to authenticate a module, it is accompanied by a security report. It consists of:

A cryptographic hash of the Public section. The Public section’s content may be altered before the protection is enabled. The cryptographic hash in the security report allows a third party to verify that this is not the case.

The layout of the module. An attacker can compromise the protection request, by issuing a \texttt{mod\_create} call with, for example, a smaller size of the Secret section. Doing so causes the module to write confidential data to unprotected memory. Using the security report and the result of a \texttt{mod\_layout} call, the expected and actual sizes of the sections can be compared when authenticating a module.

A serial and version number. Using the authentication mechanism, modules can be updated easily. The serial number is used to link together different versions of the same module. Note that the cryptographic hash of the Public section can’t be used to achieve this, since it will differ between versions. The version number is used to avoid reuse of old, potentially vulnerable, versions of a module.

Cryptographic signature by the issuer. The security report is signed by its issuer. Each protected module places trust in a number of certificate authorities (CAs), allowing verification of this signature. Because of this signature, the report can be placed in unprotected memory. In our implementation, the security report is placed right after the Secret section.

One-way authentication. To illustrate a one-way authenticated call to a module, we consider a module that generates cryptographic random numbers, which we’ll call \textit{SecureRandom}. It provides these numbers to a \textit{Client} module, as shown in Figure 3a.

In the first step, the Client uses the \texttt{mod\_layout} operation to locate the SecureRandom module’s security report. The Client then verifies the signature on the security report and the hash of the Public section, compares the actual layout with the one in the security report and checks the serial and version number.
Next, the Client calls the SecureRandom module by jumping to an entry point. All parameters, as well as the return address, are passed in registers, contrary to a regular function call on the x86 platform. The return address cannot be passed on the stack as with regular function calls, since modules are not allowed to access each other’s stack.

Finally, the SecureRandom module generates the random number and uses the `mod_return` operation to jump to the given return address. The Client needs to add the return point to its list of possible entry points, otherwise the jump to it will not be allowed by our kernel. To ensure correct control flow, the Client needs to use its Secret section to indicate that SecureRandom was called, disallowing requests to the return entry point if this is not the case.

The efficiency of this process is increased significantly in subsequent calls to the same module, by using the `mod_test` operation. By passing the id of the SecureRandom module, the kernel ensures the client that the same instance of the module is accessed and there is no need to recheck the security report.

**Two-way authentication.** Two-way authentication is very similar to one-way authentication, essentially performing the latter twice. Assume we have two modules, a Client and a Server, that wish to communicate securely, as shown in Figure 3b.

First, the Client module authenticates the Server in the same way as with one-way authentication. It then jumps to the Server module’s desired entry point, passing the return address as before. The Server uses the provided return address in an `mod_layout` call to locate the Client module’s security report. If the security report is verified successfully, the Server can securely pass confidential information to the Client.
4 Implementation in the Linux Kernel

The main contribution of this paper is an implementation of the previously described mechanism for modularization in the Linux kernel. In Section 4.1, we briefly describe the Linux virtual address space. We then describe how the access rights are maintained in the Linux kernel in Section 4.2. Finally, Section 4.3 describes the changes that were made to the operation of existing system calls.

4.1 Linux Virtual Address Space

In Linux, every process is represented by a task_struct which in turn references the mm_struct of the process. The mm_struct describes the entire process’ virtual memory, using a linked list of vm_area_struct structures. Each vm_area_struct describes a part of the virtual memory, including the user level access rights for this part of the process.

4.2 Implementation

To represent a module in the kernel, we created a new structure, called mod_struct. Each mod_struct contains the following:

- a unique identification number,
- the start and end addresses of the public and secret section,
- the last saved stack pointer.

The operations that can be applied to modules were described in the previous section. Each operation is implemented as a new system call in the Linux kernel.

Protecting the memory allocated to a module is done using the vm_flags field in the vm_area_struct. These flags define whether a virtual memory area can be read, written, executed or shared. By setting these flags to match the access rights discussed in Section 3.1, a fault occurs whenever the program violates them.

The fault handler was changed to switch the current module whenever a fault occurs at a valid entry point in the public section. Switching to a module means changing the access rights of the relevant portions of memory and switching the stack pointer to point either to the stack of the new current module or to the insecure stack.

4.3 Conflicting System Calls

Some existing system calls in the Linux kernel conflict with our policy. In this section we discuss each of these system calls and indicate how the problem was resolved.
mprotect The most obvious one is the mprotect system call, which can be used to change the access rights of a certain part of memory. To counteract this, additional flags are set on each vm_area_struct belonging to a module, indicating boundaries for what the mprotect system call may change.

mmap Some existing system calls such as mmap or mremap could be used to map a portion of a process’ virtual memory to something else. The problem here is that the secret section could be remapped to a file under control of an attacker. To counteract this, the vm_area_struct was changed to hold a reference to the mod_struct of the module to which it belongs. All system calls changing the mapping of a certain memory area were changed to set this field to NULL. The mod_test and mod_layout system calls were changed to indicate failure if this field no longer points to the correct module.

personality Each Linux process has a personality, indicating some process specific settings. One of these settings, the READ_IMPLIES_EXEC bit, indicates whether read rights to a memory region should imply that this region is executable. For protected modules, we cannot allow this, since this would make the entire public section executable at any time. Therefore, the personality of a process is changed as soon as the first module in the process is created and the personality system call was changed to disallow enabling the READ_IMPLIES_EXEC bit of the personality for processes containing modules.

fork The fork, vfork and clone system calls can be used to create a new process or a new thread. Generally, these system calls can allocate the address space to the new process in two ways.

The first possibility is that they copy the entire memory space, creating new vm_area_structs for the new process. Since we don’t want the memory belonging to protected modules to be copied to a new process, the VM_DONTCOPY flag was added to the vm_area_structs that are part of a module, which indicates that these areas should not be copied when copying the memory space.

The second possibility is that they share the entire memory space. This happens when the CLONE_VM flag is set in the system call, for example when new threads are created. To counteract this, the system call will exit with a permission error when this flag is set for a process containing protected modules.

Additionally, the mod_create system call was changed to ensure that no part of the module’s virtual memory has been marked as shared beforehand.

madvise The madvise system call can be used to change the VM_DONTCOPY flag for a section of virtual memory. Disabling this flag is not allowed for memory regions that belong to a module.
5 Evaluation

To evaluate the performance of our modified Linux kernel, we performed three types of benchmarks. First, we show that there is no performance penalty for legacy applications not using the modularization. Next, we measure the overhead induced by switching the access rights. Finally, we also benchmark an SSL-enabled web server to view the performance overhead in a real application.

We used a Dell Latitude E6510 for all our experiments. This laptop is equipped with an Intel Core i5 560M processor running at 2.67 GHz and contains 4 GiB of RAM. The operating system used is Ubuntu Server 12.04, with both an unmodified and a modified version of the Linux 3.6.0-rc5 x86_64 kernel installed.

5.1 Legacy Applications

To show that legacy applications not using the modularization technique are not impacted by our changes to the Linux kernel, we ran the SPECint 2006 benchmark on both the original and the modified version of the Linux 3.6.0-rc5 kernel. None of these tests showed a significant performance difference, with tests finishing between 0.4% faster and 0.4% slower on the modified kernel.

5.2 Microbenchmarks

To measure the overhead caused by switching the access rights, we created a microbenchmark that measures the cost of a call to a secure module and compares it to the cost of calling a regular function and calling a system call. The secure module used for this test is very simple: it simply returns by calling the `mod_return` system call as soon as it’s called. The system call and function created for this test are both very similar and just return as soon as they’re called.

<table>
<thead>
<tr>
<th>Type</th>
<th>Cycles</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure Module</td>
<td>5350823</td>
<td>900.21</td>
</tr>
<tr>
<td>System Call</td>
<td>283260</td>
<td>47.65</td>
</tr>
<tr>
<td>Function Call</td>
<td>5944</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2. Module access overhead.

Table 2 displays the results of this microbenchmark. Calling a secure module is about 900 times slower compared to calling a regular function. This overhead is attributed to the two system calls that both need to switch the access rights. Compared to calling a system call, the secure module is only 20 times slower. Due to these high costs, there is a tradeoff to be made between a low number of module transitions and small modules with additional security guarantees.
5.3 Secure Web Server

As a macrobenchmark, we modularized an SSL-enabled web server. A single module was created protecting both the server’s long time secret and the connection’s session information. Bugs in other parts of the web server no longer allow an attacker to hijack a connection or steal the server’s secret.

The secure module was built using the PolarSSL cryptographic library and a subset of the diet libc library. The server uses multiple separate processes to handle multiple connections, each with their own protected module. A simple static 74-byte page is returned to the clients over an SSL-connection protected by a 1024-bit RSA encryption key.

![Fig. 4. HTTPS-Server performance.](image)

We used the Apache Benchmark to benchmark this web server. The results are shown in Figure 4. The performance overhead tops at 10.83% and is mainly attributed to the many switches between protected and unprotected code during the SSL negotiation phase.

6 Related Work

A lot of research is done on preventing memory safety related vulnerabilities in software. However, existing practical countermeasures[3] only defend against a small set of known attacks, rather than any possible type of attack. Our goal is to modularize applications and protect the modules from each other, unless they explicitly trust each other.
A large amount of previous work focuses on isolating secure modules within applications. Many of these systems focus on protecting small portions of security-critical code.

A first approach to isolation is the use of virtualization techniques. A trusted hypervisor runs different parts of an application in different virtual machines, ensuring complete isolation between them. A first example of this approach is the Terra architecture[6], which isolates boxes of software. Another example is the Nizza architecture[7], in which security-sensitive parts of applications are extracted in AppCores and executed in a trusted process. The rest of the application runs on top of the untrusted legacy operating system. A similar approach is taken by Fides[8], where a program-counter based access control model is used to protect a Self-Protecting Module against any code residing in memory outside of the SPM. All of these systems focus on isolating a small security-critical portion of an application. As a result of the virtualization, they have a large performance overhead and are thus not suitable for complete modularization of an application.

A second approach to guaranteeing that a module can run without interference from the rest of the application is to make use of the TPM chips present in modern computers. In this approach, remote attestation is used to guarantee the correct execution of security-sensitive portions of an application to a third party. This approach is taken by McCune et al. in their Flicker architecture[9]. Flicker heavily depends on the TPM, leading to a significant performance cost. Even though this performance cost will reduce over time, as hardware performance increases, another approach based on Flicker already significantly increases the system’s performance. The TrustVisor architecture[10] uses a hypervisor and a software µTPM, simulating a TPM on the general purpose CPU. However, despite this huge performance improvement, these systems still make it a lot harder to create applications with communicating modules, since modules run in a different address space than the rest of the application.

A third and final approach to memory isolation is the use of hardware modifications. An example of this is the introduction of a program-counter based access control model in embedded systems[11], where new hardware instructions were introduced to switch between different protected modules. However, one of our design goals was to remain compatible with existing hardware and commodity operating systems.

7 Conclusion

The modularization features in modern high-level programming languages are not retained at compile-time. This paper presented a new technique for modularization consisting of compile-time changes and a modified Linux kernel. Modules are able to communicate with each other in a secure way, with minimal performance impact. Performance of legacy applications running on top of the modified kernel is not affected.
References


2. One, A.: Smashing the stack for fun and profit. Phrack 7(49) (November 1996)


Appendix B

Article (Dutch)
Huidige Trends in Kwaadaardige Software

Niels Avonds Raoul Strackx Pieter Agten Frank Piessens

20 april 2013

Het aantal computers geïnfecteerd met malware wordt wereldwijd geschat op 30%. Gebruikers zijn echter niet altijd op de hoogte van de infectie, aangezien kwaadaardige software vaak onopgemerkt probeert te blijven. Daarnaast zijn gebruikers ook niet geïnformeerd over de bestaande malware. In dit artikel leggen we uit welke soorten kwaadaardige software er allemaal bestaan, welke maatregelen er mogelijk zijn en bespreken we enkele bekende en minder bekende uitbraken van malware.[APW13, Pan13]

Malware

De naam malware is ontstaan als een samentrekking uit het Engelse malicious software (kwaadaardige software). Malware is een groepsnaam voor alle types software die je computer kunnen beschadigen of die andere kwaadaardige bedoelingen hebben. De gevolgen van de aanwezigheid van malware op een computer kunnen erg uiteenlopend zijn.

Enerzijds zijn er gevolgen die niet direct zichtbaar zijn voor de gebruiker, waarbij de malware zich zo lang mogelijk verbergt om meer schade aan te kunnenrichten. Zo kan de kwaadaardige software inlog- of bankgegevens van de gebruikers stelen. De gestolen informatie wordt dan aan de aanvaller doorgespeeld. Een tweede mogelijkheid is dat op de besmette computer een achterpoortje (backdoor) opgezet wordt. De computer wacht dan op instructies van de aanvaler. In dit geval zegt men dat de computer een zombie computer is onder de controle van de aanvaler en deel uitmaakt van een botnet.

Een botnet bestaat uit een aantal computers die samenwerken om een gemeenschappelijk doel te bereiken. Een aanvaler die een groot aantal computers kan bemsten kan deze laten samenwerken, bijvoorbeeld om grote hoeveelheden spam emails te versturen, of om een Distributed Denial of Service (DDoS) aanval uit te voeren. Bij een DDoS aanval verstoren al die computers tegelijkertijd een heleboel nutteloze data over het internet naar één server die het doel van de aanval is. Die server zal dan zoveel werk hebben met het verwerken van de nutteloze data, dat hij niet meer in staat is om tijdig te antwoorden op geldige aanvragen van gewone gebruikers.[McD11, McD09]

Anderzijds kan malware schade toebrengen die wel direct zichtbaar is voor de gebruiker. Dit kan gaan van het aanzienlijk vertragen van de computer, over het wissen van gegevens en openen van pop-up schermen tot het volledig doen vastlopen van de computer of het veroorzaken van een Blue Screen of Death (BSOD) (Figuur 1).

Soorten Malware

Zoals gewone software, kan malware ook een heeleboel verschillende functies uitvoeren. In wat volgt, bespreken we de bekende en minder bekende soorten malware.

Virus

Het computervirus is wellicht de bekendste vorm van malware. Het belangrijkste kenmerk van een virus is dat het gaat om software die zichzelf verder verspreidt.
computer, maar kunnen dit enkel als gevolg van menselijke acties, zoals het versturen van een email met een geïnfecteerd bestand als bijlage.[Coh87]

Sommige virussen hebben als enige doel om zichzelf te verspreiden. De meesten hebben echter ook andere kwaadaardige doelen, zoals beschreven in de vorige sectie.

**Worm**

Een worm is erg gelijkaardig aan een computervirus. Het gaat hier ook om kwaadaardige software die zichzelf verspreidt. De manier waarop een worm zichzelf voortplant is echter fundamenteel verschillend. Een worm slaagt er in om andere computers te besmetten zonder interactie van de gebruiker. Dit doen ze door veiligheidslekken in het besturingssysteem of in één van de programma’s op het doelwit uit te buiten. Een tweede verschil met virussen is dat een worm wel op zichzelf bestaat en dus geen ander programma nodig heeft om zichzelf aan vast te hechten.

Een computer die met het internet verbonden is kan dus geïnfecteerd worden door een computerworm zonder dat de gebruiker hier iets voor doet. Bijgevolg kunnen wormen zich snel verspreiden en een groot aantal slachtoffers maken in korte tijd. De Myspaceworm Samy is wellicht de snelst verspreidende computervirus en infecteerde 1 miljoen Myspace profielen slechts 24 uur na het uitbreken ervan.[Gro07]

**Trojaans Paard**

Een Trojaans paard is een soort malware die zichzelf voordoet als software die een nuttige functie verricht, maar eigenlijk ongeoorloofde toegang tot de besmette computer mogelijk maakt. De naam van dit type malware is een verwijzing naar de Griekse mythologie, aangezien het zich onschuldig voordoet, net zoals het paard van Troje. Trojaanse paarden verspreiden zichzelf dus niet automatisch zoals een virus of een worm.

Een gebruiker wordt overtuigd om een stuk software te installeren dat geadverteerd wordt om erg nuttig te zijn. Dit is een vorm van *Social Engineering*, het manipuleren van mensen om bepaalde acties uit te voeren die tot een besmetting leiden.

**Ransomware**

De term ransomware is afkomstig van het Engelse *ransom*, hetgeen letterlijk *losgeld* betekent. Ransomware gijzelt de geïnfecteerde computer of bestanden op die computer en vraagt de gebruiker om losgeld in ruil voor de vrijgave ervan.

Figuur 2: Ransomeware gijzelt de computer en vraagt de gebruiker om een som geld te betalen om de computer opnieuw te bevrijden. In dit geval doet de software alsof hij afkomstig is van de autoriteiten.[Pan13]
het hele beeldscherm gevuld met een waarschuwing vergelijkbaar met het scherm getoond in Figuur 2. Het scherm doet alsof het afkomstig is van de lokale autoriteiten en meldt dat er illegale activiteiten gedetecteerd zijn op de computer. De gebruiker kan zijn computer pas opnieuw gebruiken nadat hij de boete heeft betaald.

Een ander type ransomware gijzelt enkel persoonlijke bestanden van de gebruiker. Hiervoor wordt gebruik gemaakt van publieke sleutel cryptografie, een techniek om data te beveiligen tegen ongeoorloofde toegang. De bestanden worden door de malware versleuteld zodat ze onbruikbaar worden. Daarna krijgt de gebruiker een email waarin losgeld wordt gevraagd voor het vrijgeven van de gebruikte cryptografische sleutel. Enkel na het betalen van dit losgeld, worden de bestanden dus opnieuw leesbaar.[Pan13, OM12]

**Rootkit**

De term rootkit wordt geassocieerd met software die zichzelf probeert te verbergen. Infectie van een computer met een rootkit kan op twee manieren. Enerzijds kan de rootkit deel uitmaken van een groter softwarerepакket. Anderzijds kan een aanvaller een rootkit installeren na het uitbuiten van een kwetsbaarheid. Rootkits op zich hebben niet per se kwaadaardige doeleinden, maar ze kunnen wel kwaadaardige activiteiten verborgen voor de gebruiker. Merk op dat virussen, wormen en Trojaanse paarden ook rootkits kunnen zijn, indien ze gebruik maken van technieken om zichzelf te verbergen.[McD11]

Rootkits kunnen heel veel technieken gebruiken. Een veelgebruikte techniek is het verbergen van bestanden zodat de gebruiker of de antivirussoftware ze niet kan opsporen of verwijderen. Anderzijds kan het voor malware ook nuttig zijn om een proces te verborgen. De rootkit zal er dan voor zorgen dat het kwaadaardige proces niet zichtbaar is. Het proces zal uitgevoerd worden op de computer, maar zal niet voorkomen in de lijst met processen in het Windows taakbeheer.[McA06]

**Adware en Spyware**

De term adware kan enerzijds gebruikt worden om te verwijzen naar perfect legale programma’s waarin advertenties worden getoond om inkomsten te genereren voor de ontwikkelaars. Het gaat hier dan om software die door de gebruiker bewust werd geïnstalleerd. Vaak bestaat er ook een optie om de advertenties tegen betaling te verwijderen uit de software. Van zodra de applicatie die de advertenties gebeurt verwijderd wordt, zullen ook de advertenties zelf verdwijnen. Zolang het enkel bij tonen van advertenties blijft, vormt dit soort software geen bedreiging voor de gebruiker en wordt het dus ook niet gezien als malware.

In de context van malware beschrijft de term echter ook een vorm van spyware. Spyware is software die informatie over een gebruiker en over zijn gedrag verzamelt en doorspeelt aan een derde partij, zonder medeweten van de gebruiker. Bij adware wordt de verzamelde informatie dan gebruikt om advertenties aangepast aan dit gedrag te tonen. Dit is duidelijk wel een vorm van malware die de privacy van een gebruiker schaadt. De lijn tussen kwaadaardig en niet kwaadaardig wordt hier dus bepaald door het al dan niet doorsturen van informatie over de gebruiker.[McA05]

**Huidige Status**

Het aantal computers dat wereldwijd geïnfecteerd is wordt geschat op ongeveer 30%. Figuur 3 toont de verdeling van de types infecties uit 2012. Het grootste aantal infecties zijn Trojaanse paarden, met 76% van alle infecties uit 2012, een stijging tegenover de 66% uit 2011.[APW13, Pan13]

Maatregelen


**Preventie**

Malware maakt vaak gebruik van fouten of bugs in software om initiële toegang te verkrijgen tot een computer. Er zijn dan ook al een hele reeks maatregelen ontwikkeld om zowel het aantal bugs als de mogelijke gevolgen van bugs te beperken, of om het veel moeilijker te maken voor aanvallers om bugs uit te buiten. We beschrijven eerst een van de meest gebruikte soorten fouten in computerprogramma’s: een *buffer overflow* bug.

---

Buffer Overflow Kwetsbaarheden

Wanneer een gebruiker invoer geeft aan een computerprogramma, bijvoorbeeld door tekst te typen in een tekstvak, moet die invoer ergens opgeslagen worden in het geheugen van de computer. De locatie waar die data wordt opgeslagen wordt een buffer genoemd. De programmeur moet hiervoor dus voldoende plaats in het geheugen reserveren. Indien de programmeur onvoldoende ruimte voorzielt, betekent dit dat de gebruiker in staat is om voorbij de gereserveerde ruimte in het geheugen te schrijven. Je kan dit vergelijken met een papieren invulformulier waarbij er onvoldoende plaats is om je achternaam in te vullen. De resterende karakters van je achternaam zullen dan in het vak verschijnen waar eigenlijk je voornaam moest staan.

De aanvaller is dus in staat om meer geheugen aan te passen dan eigenlijk de bedoeling was. Dit geheugen bevat echter ook de instructies die bepalen wat de werking van het programma is. Een aanvaller met voldoende technische kennis, kan zijn invloed aan het computerprogramma dus zo creëren dat het programma een ander werking krijgt en eenvoudigweg de instructies van de aanvaller uitvoert. Op die manier krijgt een aanvaller de mogelijkheid om te doen met het programma en met de computer wat hij maar wil. Hij kan bijvoorbeeld wachtwoorden die in het geheugen van het programma zitten uitlezen. Anderzijds heeft hij ook de mogelijkheid om bijvoorbeeld malware te installeren.[One96]

Technische maatregelen

Omdat buffer overflow kwetsbaarheden zo vaak uitgebuit worden door aanvallers, zijn er een reeks maatregelen ontwikkeld om de gevolgen hiervan in te perken. Een eerste voorbeeld is de introductie van *stack canaries*. Op cruciale plaatsen in het geheugen wordt ruimte gereserveerd waar een willekeurige waarde wordt geplaatst. Op kritieke punten tijdens de uitvoering van het programma wordt dan gecontroleerd of deze waarde nog correct is. Indien dit niet meer zo is, weet de computer dat er in het geheugen geschreven werd op locaties waar het niet de bedoeling was. Het programma wordt dan onmiddellijk beëindigd. De naam stack canaries is een verwijzing naar het gebruik van canaries in koolmijnen om giftige gassen te detecteren.[CPM+98]

Deze maatregel kan echter slechts bepaalde delen van het geheugen beschermen, namelijk het stuk dat het vaakst aangepast wordt bij een aanval. Een tweede maatregel, *Address Space Layout Randomization*, maakt het aanzienlijk lastiger voor aanvallers om het geheugen aan te passen. Telkens wanneer een programma gestart wordt, worden alle buffers en andere delen van het programma op willekeurige plaatsen in het geheugen gezet. Het wordt voor een aanvaller dan veel moeilijker om selectief bepaalde locaties in het geheugen te overschrijven, hetgeen een geslaagde aanval veel ingewikkelder maakt.[SPP+04]

Andere Maatregelen

Naast deze technische maatregelen, zijn er ook nog andere mogelijkheden om aanvallen te beperken. Een eerste mogelijkheid is het gebruik van een veilige taal waarbij het vaakst aangepast wordt bij een aanval. Een tweede maatregel, *Address Space Layout Randomization*, maakt het aanzienlijk lastiger voor aanvallers om het geheugen aan te passen. Telkens wanneer een programma gestart wordt, worden alle buffers en andere delen van het programma op willekeurige plaatsen in het geheugen gezet. Het wordt voor een aanvaller dan veel moeilijker om selectief bepaalde locaties in het geheugen te overschrijven, hetgeen een geslaagde aanval veel ingewikkelder maakt.[SPP+04]

Een tweede optie is het bewijzen van de correctheid van je programma. Als gevolg van recent onderzoek zijn er tools ontwikkeld die het mogelijk maken om te bewijzen dat je programma geen geheugen aanpast dat het niet zou mogen aanpassen. Het nadeel van dit soort tools is dat het extra tijd vraagt van de programmeur en dat het momenteel enkel goed werkt voor relatief kleine programma’s.[JSP10]

Tot slot wordt er ook onderzoek gedaan naar hoe bepaalde delen van programma’s kunnen beschermen worden. Zo blijven de mogelijkheden van een aanvaller, zelfs na een geslaagde aanval, nog vrij beperkt. In deze oplossing wordt het geheugen van een computerprogramma opgesplitst in verschillende modules en heeft elk van die modules slechts beperkte rechten. Een aanvaller die erin slaagt om een kwetsbaarheid in een module uit te buiten, verkrijgt slechts dezelfde rechten als die module zelf en kan niet aan het geheugen van de andere modules. Dit soort isolatie zorgt ervoor dat een aanvaller veel beperktere toegang krijgt tot de computer.[ASAP13]

Detectie

De tweede lijn van defensie tegen malware bestaat uit het detecteren ervan nadat hij is geïnstalleerd. Dit is exact waar antivirussoftware voor dient. De manier waarop antivirussoftware malware detecteert kan sterk verschillen van product tot product.[GF11]

De oudste techniek maakt gebruik van *virus signatures*. Hierbij wordt de code waaruit een virus is opgebouwd gebruikt als teken voor de aanwezigheid ervan. Een virusscanner bevat dus een database van virussen en de code waaruit ze zijn opgebouwd. Met deze techniek kunnen echter enkel reeds bekende virussen opgespoord worden.

Een tweede techniek werkt op basis van heuristieken. Het antivirusprogramma bevat een aantal regels die typische malwarekarakteristieken beschrijven. Elk van die regels wordt dan toegepast op een bestand. Op basis van overeenkomsten met bepaalde regels wordt een score toegekend aan het bestand. Indien die score te hoog is, wordt het bestand behandeld als mogelijk besmet.[HL11]

Een laatste techniek die vaak gebruikt wordt is het uitvoeren van een verdachte programma in een aparte geïsoleerde omgeving, een zogenaamde *sandbox*. Uit het resultaat daarvan kan dan bepaald worden of het
om een kwaardaardig programma gaat of niet.

Bekende Uitbraken

Ondanks al deze maatregelen, blijven grootschalige aanvallen een realiteit. Hieronder worden een aantal van de grotere en meer bekende cyberraanvallen en uitbraken van malware aangehaald.

Stuxnet Worm

De Stuxnet worm is een erg complex stuk malware dat ontworpen is om een gerichte aanval uit te voeren. De worm verspreidde zich initiëel via verschillende bekende en onbekende kwetsbaarheden in onder andere het Windows besturingssysteem en was speciaal ontworpen om niet ontdekt te worden. Hij bevatte hiervoor een Windows rootkit die zijn bestanden verbergt. Uit een analyse van de worm werd echter afgeleid dat deze Windows computers slechts tussenstappen waren om zijn uiteindelijke doel te bereiken.

Het uiteindelijke doel van de aanval waren industriële controlesystemen van Siemens, onder andere gebruikt in nucleaire installaties. De controllers (PLC’s) van deze systemen worden geprogrammeerd met behulp van een Windows computer en Siemens software, Step 7 genaamd. Van zodra Stuxnet er in slaagde om deze computer te infecteren, werd de Step 7 software aangetast en nam hij controle over de communicatie tussen de Windows computer en de PLC.

Het probleem met de software die Sony BMG meeleverde, was dat hij ook geïnstalleerd werd als de gebruiker de License Agreement weigerde. Daarnaast maakte de software gebruik van Rootkit technieken om zichzelf te verbergen en is hij erg moeilijk te verwijderen. Tot slot bevatte de software zelf ook kwetsbaarheden die later door andere virussen en wormen werden uitgebuit.

Figuur 4: Infecties met Stuxnet per land: 60% van de infecties bevindt zich in Iran, hetgeen erop wijst dat het uiteindelijke doel van de worm waarschijnlijk een Iranees controlesysteem was. [FOMC11]

DDoS Aanval op Zuid-Korea

Tot slot bespreken we een aanval die recent nog extra media-aandacht kreeg. Het betreft hier een Distributed Denial of Service (DDoS) aanval tegen een aantal populaire Zuid-Koreaanse websites, zoals de website van de president, van het ministerie van defensie en van de Shinhan Bank.

Zoals eerder al uitgelegd, wordt er voor dit soort aanval gebruik gemaakt van een groot aantal zombie computers die samen een botnet vormen. Elk van die computers stuurt zelf een aantal aanvragen naar het doelwit. Het doelwit kan de overvloed aan verkeer niet meer aan en gaat onderuit.

De aanval dateert al uit 2009, maar recent kwam aan het licht dat ook Belgische computers hebben deelgenomen aan de aanval. Dit is het gevolg van de manier waarop de malware geïnstalleerd op de computers uit het botnet, zich verspreidde. Het toen recente overlijden van Michael Jackson zorgde voor grote populariteit voor de site eternalmoonwalk.com die als eerbetoon werd opgericht door Studio Brussel. De server waarop deze werd gehost was echter besmet. Er werd gebruik gemaakt van een bug in de populaire Flash-plugin om malware te installeren op de computers van bezoekers aan de site. Bijgevolg namen minstens enkele honderden Belgische computers deel aan de DDoS aanval op Zuid-Korea. [Van13]
Referenties


Bibliography


[24] J. Koetsier. Android up 13%, iOS down 7%, BlackBerry down 81% ... and Windows Phone up a massive 52%. Venturebeat, April 2013.


Fiche masterproef

Student: Niels Avonds

Titel: Implementation of a State-of-the-Art Security Architecture in the Linux Kernel

Nederlandse titel: Implementatie van een State-of-the-Art Beveiligingsarchitectuur in de Linux Kernel

UDC: 621.3

Korte inhoud:
Estimates show that around 30% of the computers worldwide is infected with malware. Malware spreads quickly and automatically by exploiting low-level vulnerabilities in user-level applications, such as buffer overflows. This type of attack leads to the ability to execute arbitrary instructions in a process’s address space. The attacker not only gains full access to the application’s secrets, but is also able to request services of the operating system with the same access rights as the user running the exploited process.

Existing countermeasures focus on prevention of specific attack vectors, but crafty attackers continuously develop new exploits. Consequently, recent research focuses on damage reduction instead, assuming that successful attacks are inevitable. In one research track, security-critical functions are protected by placing them in separate modules that are fully isolated from the rest of the application.

This master’s thesis introduces modifications to the Linux kernel that allow applications to be divided into such protected modules. Rather than focusing on the isolation of security-critical functions alone, the Linux kernel modifications enable completely modularised applications. By guaranteeing that an attacker able to compromise a single module cannot extend his reach to the other modules, his power over the rest of the application and its secrets is limited. The attacker’s power with respect to the operating system is reduced significantly as well, by selectively disabling system calls for each module.

In this thesis, the design of these modifications is discussed and a prototype implementation is described and evaluated. The modified kernel retains compatibility with existing hardware and software. Legacy applications and protected applications thus run side by side, on top of the same operating system kernel. The performance of legacy applications is shown to be unaffected while protected applications show considerable security improvements with an acceptable performance overhead.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdspecialisatie Veilige software

Promotor: Prof. dr. ir. Frank Piessens
Assessoren: Dr. ir. L. Desmet
Dr. ir. J. C. Gomez Carranza
Begeleiders: Ir. P. Agten
R. Strackx