Technical Report 2014-001
An Approach to Stack Overflow Counter-Measures Using Kernel Properties

Benjamin Teissier and Stefan D. Bruda
Department of Computer Science
Bishop's University
2600 College St
Sherbrooke, Quebec J1M 1Z7, Canada
teissier@cs.ubishops.ca, stefan@bruda.ca

20 May 2014

Abstract

Our work contributes to the investigation of buffer overflows by finding a more accurate way to prevent the exploitation of buffer overflows. Our working platform is the GNU/Linux family of operating systems. Our work is at the highest privilege levels and in the safest part of a GNU/Linux system, namely the kernel. We provide a system that allows the kernel to detect overflows and prevent their exploitation. Our system allows the kernel to inject at launch time some (minimal) code into the binary being run, and subsequently use this code to monitor the execution of that program with respect to its stack use, thus detecting stack overflows. The system stands alone in the sense that it does not need any hardware support; it also works on any program, no matter how that program was conceived or compiled. Beside the theoretical concepts we also present a proof-of-concept patch to the kernel supporting our idea. Overall we effectively show that guarding against buffer overflows at run time is not only possible but also feasible, and we also take the first steps toward implementing such a defense.

1 Introduction

IT security is a complex and important field, having a long history. It became recently a subject of general concern for the main public, for governments, and for private companies alike. Indeed, the latest news involving stuxnet, flame, and so many other viruses made clear to everybody that privilege escalation and security threats in general deserve a greater attention.

One of the oldest issues in this field is buffer overflow. The range of buffer overflow exploitations in particular is large, ranging from data leaks to taking over a complete computing system. These are all dangerous, and most of the time the possibilities of exploitation are only limited by the skills of the attacking hacker.

This problem appeared early in the history of computers. Indeed, buffer overflow is first mentioned as early as 1972 [1], and the first documentation of a hostile exploitation was written in 1988 [21]. However, as can be seen during events such as the NDH 2k11 "old hacking" conference [12], the exploitation of buffer overflows were present earlier, but some times documented privately and often not documented at all. A few worms are known to use buffer overflows for exploitation, including "SQL Slammer" [13] and "Code Red" [5]. They infected a substantial number of computers and servers, compromising data, computing time and the security of the entire system.

In its early history the exploitation of buffer overflows was reserved for the elite of hackers. It was often learned within hacker communities, and further developed and refined in the 80's within the most popular such communities including the German Chaos Computer Club (since1980) and the Cult of the Dead Cow (since 1983), but also within convention like the DEF CON (since 1993), the Chaos Communication Congress (since 1984), the Chaos Communication Camp (since 2003), the Nuit Du Hack (since 2003), etc. However, the problem got out of communities and conventions and became available for the non-elite people with the famous paper “Smashing the stack for fun and profit” [14], written by Elias Levy (alias Aleph One) and published in the Phrack webzine in 1996. With this paper, the buffer overflow was popularized; additionally, a good and complete “how to” was made available to everyone. All of a sudden buffer overflow exploitation was made available to everybody, even to people without strong knowledge.

Buffer overflow is an old issue but at the same time it is a current and acute problem. We can see this by looking at the exploit-db.com Web site, a giant archive of exploits and vulnerable software: more than 100 pages (with some 20 articles per page) are dedicated to buffer overflows.

This family of bugs is manifest in programming languages that give liberty to the programmer, particularly the C and the C++ languages. This does not however limit the extent of the problem, as C has been one of the most popular language since 1970 [25] and C++ is not this far behind. Such languages leave the task of managing memory (and thus securing it) at the latitude of the developer.

Computer security is a huge issue in our ultra-connected society. People do everything with a computer, from buying stuff on-line to making and watching personal movies, so any failure is a notable event. The security issue is even more of a problem for banks, governments, nuclear power stations, energy management companies, and armed forces. They all operate in a “critical environment” that is, an environment with some strict constraints on time and/or space, whose failure has wide and unacceptable implications, ranging from financial loss to even loss of life.

Old program running on on old computers often cannot ever be patched for bugs. The reasons range from lack of expertise to inaccessibility of the course code. Our goal is therefore to find something to stop the exploitation of stack overflow for all the programs instead of cleaning all the pieces of code of stack overflows one by one.

Our thesis is therefore that a solution for stack overflow implemented at the level of the kernel of an operating system and not involving recompilation or supplementary components is possible.

The purpose of our work is thus to find a solution for the stack overflow problem. Our solution will be at the level of the kernel of the operating system, thus addressing the problem automatically for all the programs running on a machine. We work on the Linux kernel, since this kernel is open source and thus we have access to its source code. Moreover, this kernel is modifiable by the computing community and it will be possible to have our patch accepted in the official kernel source (which is impossible for close-source kernels).

Overall we propose a patch to the Linux kernel that will mitigate the stack overflow issue.

2 Buffer Overflow Preliminaries

In this section we address the buffer overflow in general and then we particularize the discussion to stack overflow.

The buffer overflow is a bug originating from a failure of the developer. It consists in a program wanting to write into a variable (e.g., array) but ending up writing outside the respective variable. Very often, a buffer overflow causes an overwrite on another part of the program's memory. This results in an unpredictable behavior of the system, ranging from erroneous results of the program with the bug to a system crash.

Often buffer overflows can be exploited by malicious entities. Here is an example of a simple buffer overflow and its exploitation. First, we disable ASLR[1]:

[1] ASLR stands for Address Space Layout Randomization and is a technique that randomizes the memory space in order to reduce the likelihood of buffer overflow exploitation. It will be described later.

```
root@bt:~# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

We then create the following, faulty piece of C code. The failure is caused by the strcpy function. The function copies a string source into a buffer, but if the string is larger than the buffer then the function will also overwrite a part of memory outside of the destination. The effect is unpredictable.

```
root@bt:~# cat vuln.c

#include <stdio.h>
#include <stdlib.h>

void mainfunc(const char *s) {
    int i = 1;
    char buffer[12];
    strcpy(buffer,s);
    printf("i = \%u\n",i);
}

int main(int argc, char **argv) {
    if(argc != 2) {
        exit(EXIT_FAILURE);
    }
    mainfunc(argv[1]);
    return EXIT_SUCCESS;
}
```

We compile this program:

```
root@bt:~# gcc vuln.c -w -O0 -ggdb -std=c99 -static -D_FORTIFY_SOURCE=0 \
  -fno-pie -Wno-format -Wno-format-security -fno-stack-protector -z norelro \
  -z execstack -o vuln
```

and then look for problems using an increasingly long argument:

```
root@bt:~#./vuln abc
i = 1
root@bt:~#./vuln AAAAAAAAAAA
i = 1
root@bt:~#./vuln AAAAAAAAAAAAAAAAAAAAAAA
i = 1094795585
root@bt:~#./vuln AAAAAAAAAAAAAAAAAAAAAAAAAA
i = 1094795585
Segmentation fault
```

We also create a script called SC and put it in an environment variable. We add 10 “\x90” (NOP) instructions, as it is easier to focus on a larger memory area than on a unique address. The following script is a shell code (meaning that it will launch a shell) written directly in machine language:

```
root@bt:~# export SC=\$(python -c ’print "\x90"*10+
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e
\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"’)
```

With a simple program called getvarenv we find the address of SC into the memory:

```
root@bt:~#./getvarenv SC
SC is located at 0xbffffe32
```

Then we can smash the stack putting in it the address of our shell code:

```
root@bt:~# ./vuln \$(python -c ’print "\x32\xfe\xff\xbf"*8’)
i = 3221225010
sh-4.1\#
```

The reason for our success is that the shell code is first placed on the stack. The overflow produced by strcpy will overwrite the EIP value pushed on the stack and so change the program flow, so that the current function will jump to our shell code rather than returning. The program opens a shell prompt as directed by our shell code. We just took over the program and we thus have a full-blown shell running with the user id of the initial program. If the program had root permissions to begin with, then we can do anything we want with the machine.

A *stack overflow* is a buffer overflow happening on the stack. The attack vector is a fixed size array. Often caused by the use by the strcpy() (which does not account for array size), the stack overflow is the most popular and the easiest buffer overflow to detect, patch, and then exploit. The attack targets the EIP register and takes over the program. Indeed, the register EIP is the instruction pointer register, which stores the next instruction to be executed. When the program calls a function, it will store the actual value of EIP on the stack to restore it at the end of the function. In the example above the EIP value stored on the stack is overwritten and replaced by the address of the shell code, which causes its execution.

Other overflows include integer overflow and the heap overflow. This first attacks integer variables and attempts to increase their values beyond their capacity (so that they roll over), while the second target the dynamic arrays created using the malloc() family of functions (malloc(), calloc(), realloc()).

3 Previous Work

The *Nx-bit* stands for “Never execute” and is a technique that identifies two different parts of the memory [15]. One part contains data, and this segment can be overwritten. The other part contains instructions and also can not be overwritten. The Nx-bit creates a distinction in the writing permissions between the memory initialized at the start of the program (instructions, locked in writing), and the memory initialized and modified on the fly (data, unlocked in writing).

The Nx-bit counters shell code injected in the memory. Indeed, the shell code can still be loaded in the memory of the program, the EIP register can still be overwritten to point to the beginning of the shell code, but because of the Nx-bit the shell code will still not be executed. Indeed, the stack is protected for execution, so the program will crash instead of executing malicious code. This protection can however be bypassed simply with a return-to-libc attack [4].

ASLR stands for Address Space Layout Randomization and is a technique that randomizes the memory space of a program [2]. Generally the position of the stack, the heap and third-party libraries are all randomized. This randomization introduces a component of chance: The virtual addresses within the randomized space will change at each start of the program, which counters attacks based on fixed structures.

This security measure is pretty effective, but the automatization of the brute force or the insertion of Nop instructions reduce the effectiveness of randomization. Indeed, the brute force will test quickly the program, and if the program crashes or cannot be exploited because EIP points to a bad address or an illegal instruction (data for example), the script will reload immediately the program and test again and again [22]. The Nop instruction will allow a large landing place for EIP, thus simplifying the process.

A *stack canary* is a defined value that is put just after pushing the EIP value on the stack. It can be checked before popping the EIP at the end of the function [7]. This method prevents the modification of EIP and also the take-over of the program by a third party. Stack canaries do not add code and are pretty good, but they can nonetheless be bypassed [3, 17]. The bypass of stack canaries will be studied in my project.

The *64-bit architecture* changes some interesting details. For example, the increased number of registers allows for function parameters (if they are less than 6) to be transmitted via registers rather than through the stack. The fact that an address is not represented using 64 rather than 32 bits increases the pool of valid address considerably, and so the blind stack smashing becomes harder. Harder however does not mean impossible, so ultimately a 64-bit machine is just as vulnerable as a 32-bit machine.

The mmap() function (for memory map) is a POSIX system call. This function creates in memory clones of files or devices. The *randomization of the mmap() base* introduce some chance to avoid attacks such as return-to-libc. The return-to-libc attack uses the shared library libc which is linked to every C program. This attack is useful against the NX-bit defense, for indeed the attack does not need any injection of code, using instead the powerful system() function included in libc. With randomization, the addresses of the linked shared libraries will be randomized, and so the start of the libc into the memory is no longer fixed. This will slow down the attack but a brute force approach is nonetheless able to overcome this protection.

3.1 Most Recent Research on the Subject

Defending against buffer overflow vulnerabilities [20]: This work summarizes different ways to take over a program using overflow attacks. It also outlines defenses at the developer level, consisting essentially in secure coding practice. Another defense presented here is at the compilation level and consists in the introduction of Stack Guards or Return Address Defenders, both of which will provide a protection upon compilation. A description of dynamic code analysis and its combination with static analysis is presented, together with network-based instrumentation, where the network data is compared with signatures from older attacks. The paper finishes with the presentation of detection tools before or after the compilation. In the first case, the source code is analyzed, while in the second case the binary code is analyzed. The author essentially explains why the creation of a method to prevent "buffer overflows" is impossible due to the different attack methods. The problem of the methods presented in the paper is that either a compilation, or detection before compilation are needed. Without access to source code (and permission to compile and install software), these methods cannot be applied.

Security protection and checking in embedded system integration against buffer overflow attacks [24]: In this work a new approach of buffer overflow defense was proposed. This approach is at the hardware and software level and proposes new assembly functions to secure the system by two methods.

The first method, "Hardware Boundary Check" offers a simple but very smart protection: When writing function is called, a parallel function runs and checks the value of the target address. If this address is equal or larger than the value of the frame pointer, the protection is engaged.

The second method consists in the rewriting of function calls and returns with the introduction of two new opcodes, namely "SCALL" and "SRET." The first one produces a signature at the point of the call, while the second one check the signature before the return and detects any modification.

This work is really interesting but does have some limitations. The first method adds new control code at each function call and return, which could be a performance issue. The second method need a third party component and specialized hardware.

Hardware/software optimization for array & pointer boundary checking against buffer overflow attacks [23]: The approach in this paper is not new, as it is based on boundary checking. The main problem with this solution remains performance: programs took two to five times more to run after the introduction of boundary checks. The idea of this work is then the implementation of a new instruction to limit the use of resources and optimize the process. The optimization of a proven method is a good idea but the main problem is the same as for other research, namely the need of third party components.

4 A Kernel Patch to Prevent Stack Overflows

The basic idea of our patch is the control of the user space eip stacking by the kernel space. Theoretically any kind of control on the software side is possible from within a kernel so such a simple and less intrusive control should be possible. We will therefore explore the Linux kernel and accomplsh said control within its possibilities and limits.

The problem of buffer overflows clearly stems from eip stacking. An adversary can modify various other variables with a buffer overflow and thus dodge any protection. Once eip stacking is exploited the situation becomes application specific and so impossible to control using a general mechanism. We therefore focus of the common root cause. We do not have a lot of solutions to this problem: we can either forbid the overwriting of eip, or check the value of the stacked eip before using it. Stack canaries work with the second approach, while our work tries to follow the first route. To prevent eip overwriting we will try to implement a solution close to the stack canaries but independent of the particular binary being run. This solution should be implemented into the operating system, just like the ASLR. The focus of this work will be the call and ret instructions, critical points of a program.

4.1 Early ideas and their problems

The first idea of the implementation was the design of a hook into the text segment of the process for each call and ret. The call instruction will stack the eip and lock for writing the respective stack part. The ret instruction will unlock the respective part and then restore the value into the eip register. Memory locking and unlocking are available with kernel functions. We were hoping that we will be able to hook into stacking and unstacking (push and pop instructions) from within the kernel, as a userspace program works with virtual addresses while the kernel is the one responsible for their translation into physical addresses. The reality is however different and the kernel has no influence on this process, so the hooking of call and ret instructions is compromised. We thus face our first problem.

Even if we can hook into push and pop we still face a memory management problem. The most obvious memory locking mechanism is page allocation. However, it is impossible to lock parts of a page, and the minimal size available for a page is 4Kib. Thus a naïve approach to eip locking we would effectively use 4Kib of memory to store a 32-bit address. Needless to say, this is extremely wasteful and so unacceptable for anything else but toy demonstrators.

Unsurprisingly, naïve solutions do not work. We had to find another approach.

4.2 Solving the problems

Hooking into stack manipulation can be done in several ways. We however believe that only one such a way is realistic. Our starting idea comes from the stack canaries protection, that adds a few instructions before the call and the ret instructions at the compilation stage. The idea is thus to inject a small piece of code in the program being run so that we can generate an interrupt and so enter into the kernel space (where everything becomes possible). The simplest and most logical mechanism is system calls. Such a system call takes two instructions, one to move the system call number into eax and one to call interrupt 80, the system calls interrupt.

System call example in assembly and binary, here `my_call`

```
mov eax, $15f     b8 60 01 00 00
int 0x80          cd 80
```

Recall however that one of our goals is for our mechanism to be implemented into the operating system rather than at compilation time. This create three new problems: One should find the process, then determine the appropriate places for the insertion of our system calls into the memory, and finally modify the text segment appropriately.

The first problem (finding the process) is easy to solve. We have a whole family of probes that can be used. The second problem (finding the insertion places) is complicated by the CISC instruction set of the processor family used in our testing (and indeed in most computers in existence today). Such an instruction set features several opcodes with essentially the same result [9]. Moreover, the CISC instruction set features variable instruction lengths, which effectively prevent the sequential parsing of the text segment to find the needed instructions. We will consider this problem in depth a bit later. We could not find a definite solution to the third and final problem (modifying the text segment). Instead we will present a workaround and we will discuss possible approaches to a definite solution in our conclusions.

Recall that memory locking and unlocking is unrealistically wasteful and so should be avoided. We eliminate the need for such operations by the use of a kernel-space eip stack. As the kernel is supposedly safe, we can safely store critical data inside. As described above, we got a system call into the kernel space before call and ret instructions. When a call instruction happens we pick up the value of eip plus the size of the following call and store it into a variable kernel side. Then the call will push on the stack the eip value, as usual. Just before the ret instruction we have a new kernel interrupt which checks the top of the stack against the value saved into the kernel. If the two values agree with each other then nothing else needs to be done and the program continues normally. If a difference is noted, then the program should be considered corrupt and appropriate action should be taken. We chose to terminate the program in such a case, though other actions can be easily implemented instead. Our framework even allows for the possibility of a ret to the kernel-stored address, thus restoring the normal return point of the current function; however we believe that such an action is not the best approach, as the program is already known to be corrupted so running it further is likely to result in errorneous (and potentially dangerous) behaviour.

All this being said, we are now ready to describe our patch to the Linux kernel and discuss in depth our choices.

4.3 Implementation

Figures 1, 2, and 3 summarize the description of our system. A detailed description is provided below and the complete implementation is presented in the appendices.

4.3.1 Kernel modifications

A few vital modifications into the Linux kernel were necessary:

- *mm/memory.c line 4154*

```
EXPORT_SYMBOL(access_remote_vm);
```

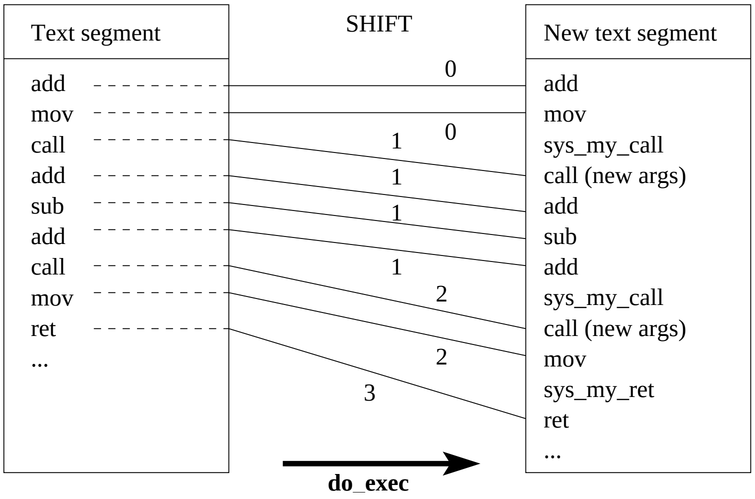

Figure 1: The do_exec module rewrites the text segment from bottom to top into a new text segment, shifting the addresses to make room for the patches (i.e., new system calls). The amount of shifting is decreased each time a patch is injected.

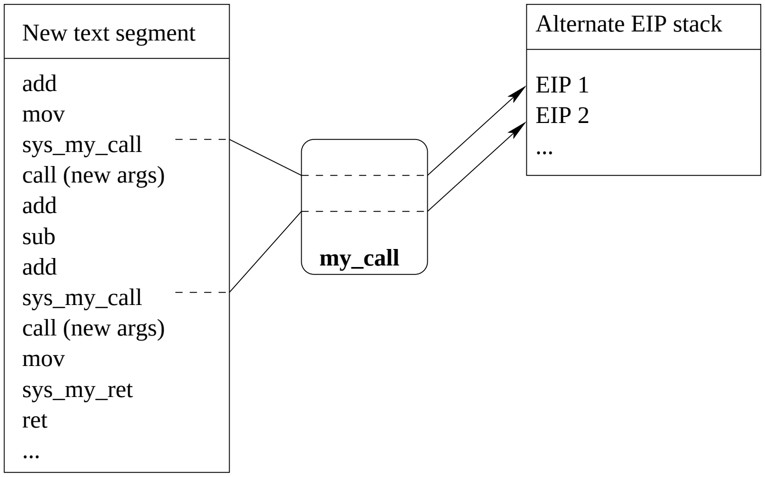

Figure 2: The my_call module probes the systemcall sys_my_call and copies the current eip of the program into the alternate EIP stack.

We allow modules to access virtual memory areas for reading and writing, which is critical to the patch.

- *include/linux/mm_types.h line 324*

```
struct Alt_stack{
    unsigned long eip;
    struct list_head mylist;
}
```

We use this structure to store the stacked eip into the kernel.

- *include/linux/mm_types.h line 444 into the declaration of mm_struct*

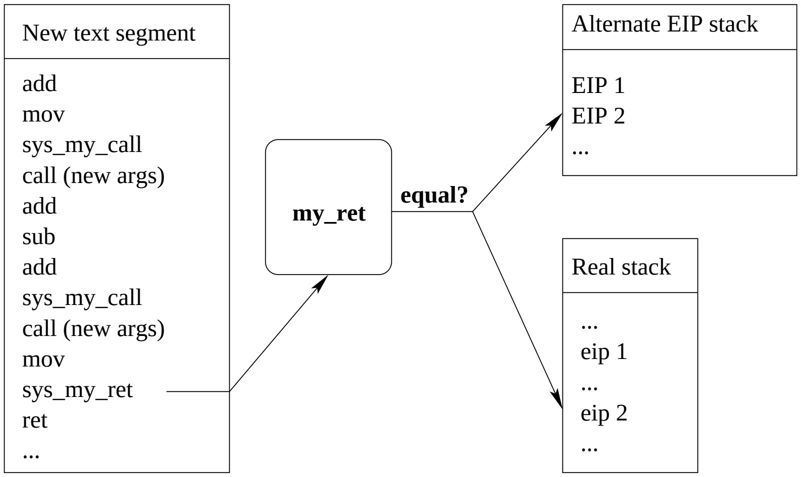

Figure 3: The my_ret module probes the systemcall sys_my_ret, check the last eip stacked into the real stack and the last eip stacked into the alternate stack. If they are different, the program is killed, otherwise the alternate eip stack is popped.

```
Struct Alt_stack alt_eip_stack;
```

We declare the new structure into mm_struct. Each process has such a structure and only userspace programs use it.

- *arch/x86/syscalls/syscall_32.tlb line 360*

```
351         i386        my_call       sys_my_call
352         i386        my_ret        sys_my_ret
```

This file contains all the system calls for the x86 32-bit architecture. We define here the number associated to our new system calls, the platform where it can be used, the name of the function called by the respective system call, and the name of the function displayed in /proc/kallsyms.

- *include/linux/syscalls.h line 902*

```
asmlinkage        void        sys_my_call(void);
asmlinkage        void        sys_my_ret(void);
```

A second declaration of the new system calls is needed in the appropriate header file.

- *kernel/my_call.c*

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/sched.h>
#include <linux/syscall.h>
asmlinkage void sys_my_call(unsigned long eip) {}
```

The system call my_call is exceedingly simple, as it is empty (does nothing), takes nothing as parameters, and returns nothing. Indeed, we use this system call just as an interrupt so that the kernel can take control in the appropriate places.

- *kernel/my_ret.c*

  ```
  #include <linux/kernel.h>
  #include <linux/init.h>
  #include <linux/sched.h>
  #include <linux/syscalls.h>
  asmlinkage void sys_my_ret(void) {}
  ```

  Except for the name of the file and the name of the function, our system call my_ret is exactly the same than my_call and serves exactly the same purpose.

- *kernel/Makefile*

  ```
  obj-y += my_call.o
  obj-y += my_ret.o
  ```

  We added our custom system calls to the compilation process.

4.3.2 Module implementation

We used several modules. Each module includes a probe and so can handle just one address (or function). The modules my_call and my_ret are also part of the user space interrupt. Our modules are as follows:

- **do_exec:** The do_exec module probes the kernel function “copy_process”. The name of the module should therefore be copy_process. However the original function being probed was do_exec and we kept the original name since the name of the module is immaterial with respect to the actual function being probed.

  This is our main module. It gets access to the program just after the respective binary is copied into memory. It then calls the disassembler via a Python script, modifies the binary into memory by injecting code and recalculating shifting. Once all this is done, this module initializes the linked list of stack eip values.

  Ndisasm and objdump are two disassemblers, which produce readable assembly code out of a binary file. They were useful for the comprehension of Linux binaries (typically ELF files) but also for the modification of binaries manually in order to test some features of our system.

  Objdump is necessary in our system given the variable length of instructions in the CISC instruction set. It is used in order to eliminate the need of a disassembler into the kernel. We found Objdump better than Ndisasm for our purposes, for indeed Objdump returns the entire address of instructions and can also isolate the text segment. Objdump is therefore used by our patch in conjunction with a small script, awk and egrep. The output give the addresses and names of the instructions of interest.

- **my_call:** The module probes the syscall my_call and so gets a chance to put the actual eip into the kernel linked list.

- **my_ret:** This module probes the syscall my_ret. It checks the last eip stacked and the last eip into the kernel linked list. The checks are performed before the ret so that potential exploitations are prevented.

The last two modules use the *probe family, a mechanism implemented in the 2.6 version of the Linux kernel by Jim Keniston, Prasanna S Panchamukhi and Masami Hiramatsu [18]. Kprobe, Jprobe and

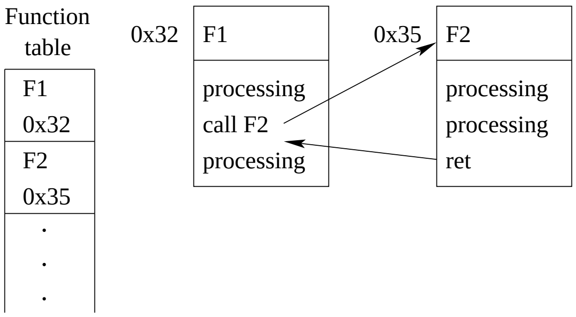

Figure 4: Normal function call: The function table includes the name of the function (top) and the address of the function into memory (bottom).

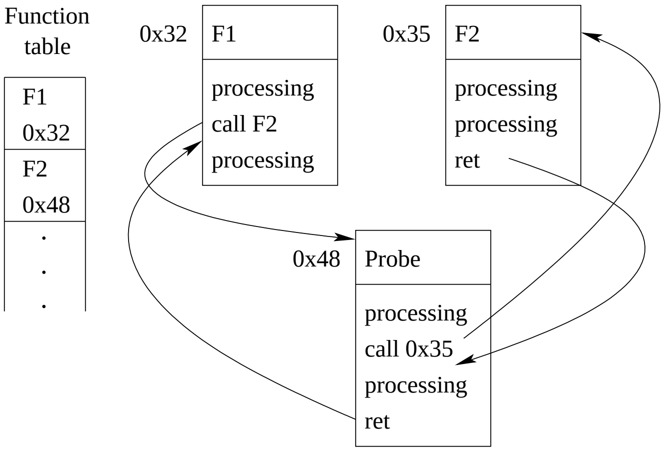

Figure 5: Probing a function: The address of F2 in the function table was changed to the address of the probe (0x48). The program flow is modified before and after the execution of F2.

Kretprobe are all members of this family. This family allows us the implementation of probes, which between other things are accessible in modules and so help with avoiding kernel modifications. The available probes are different for each mechanism. Kprobe provides a probe before and after the target function, Jprobe give us the arguments and a probe before the target function, and Kretprobe give us the return as well as a probe after the target function.

The probe family uses the hooking technique: The module using a probe for function "function_name" will change the address used to call function_name to the address of entry into that module, while the last operation of the module will cause the program to jump to the original address for function_name. When the probe is removed, the original address is restored. This is illustrated graphically in Figures 4 and 5.

4.4 Testing

Testing is fairly simple and straightforward. We provided a "safe" program and an "unsafe" one. Those two programs were tested using the techniques described in Section 2. They were then tested, first with normal protection (ASLR, NX-bit and stack canaries) and a second time without the ASLR protection in place (see Appendix I) but running under our system.

In the first test (that is, outside our system) the safe program executed normally and terminated properly. The injection of malicious code in the second program worked, as illustrated by the following output:

```
aBufferoverflow
Bufferoverflow
Bufferoverflow
Bufferoverflow
Bufferoverflow
Bufferoverflow
Bufferoverflow
Bufferoverflow
Segmentation fault
```

In our second test (under our system) we first investigated the in-memory code pre- and post-patch to ensure the correct injection of system calls, the appropriate shifting, and the correctness of calculations for the call arguments. Once all of these were found to be satisfactory, we ran the two programs. As expected, the first, correct one worked without any problem, while the second program stopped before the exploitation of the buffer overflow became possible and displayed nothing. We also used a a third program, created to test system calls. The testing programs are included into Appendices F, G, and H.

5 Discussion and shortcomings of the patch (and how to remedy them)

The patch is not perfect, for indeed this is a proof of concept that needs to be further refined to become a production system. The results of testing are encouraging and so we believe that pursuing such a production system is possible and relatively straightforward. The following shortcomings need however to be addressed for this to happen.

5.1 Cleanliness of the code

The protection of the text segment is functional but is not as elegant than we wanted. Indeed the use of the Python script and disassembler to find the call, ret, and jump instructions is ugly and not secure. The lack of security is caused by the invocation of a user-space program. Moreover, the whole process is slow. In particular objdump spends a considerable amount of time disassembling parts of the code that are not interesting to our system and putting the result into a file. This is a waste of energy but is nonetheless essential to our system, as a kernel module cannot invoke objdump directly and there is no equivalent disassembler on the kernel side. The CISC instruction set does not appear to allow any cleaner solution, but this issue should definitely be investigated in depth.

5.2 Entry point

A problem discovered during tests was related to the entry point of the program. Indeed, a program needs to know where the main function starts, but this entry point is sometimes modified by the packer. If the patch modifies and moves the entry point by injecting code before its location, we should rewrite the real entry point. This one can be easily found into the binary since it looks like this:

```
8048347:  68 30 84 04 08  push  $0x8048430
804834c:  e8 cf ff ff ff  call  8048320 <__libc_start_main@plt>
```

The modification of the entry point should be easy and look like the modification of call argument, except that here its address is absolute and not relative: first we find the old address, then we calculate the shift needed, and at the end we rewrite the new address. It should be easier to do at the end and to calculate the result as we proceed with the patching of call and ret.

5.3 Memory optimization

The part of the memory used by the first module that reads the binary is fixed and not optimized. It is faster to execute and it avoids some troubles with the schedule, but eating so much memory is bad, especially on the kernel side. The kernel obviously does not have an unlimited memory, for one thing because the computer does not either but most significantly because the address space of the kernel is limited [26]. If the module use 100kb, and 100 applications run together, the total of the memory used by the module at this time will be 10mb. This is still not that much, but what would happen if the program ran is a huge application such that libreoffice or gimp? The possibility of exhausting the kernel memory space is real and therefore memory management for the module should take an important place in the development of the production version.

Reducing the memory consumption requires substantial work. The module needs a lot of information to read the binary: location of call, ret, jump and the type of instruction (a simple integer) plus two addresses of the beginning and the end of the "real" text segment. Those two addresses should not be optimized as there are two 32-bit addresses and are independent of the application size. For the rest, we believe that the use of an "on the fly" debbuging-patcher will resolve most memory problems.

5.4 Modification of the text segment missing

We do not modify any jump or call before the main function. A call before the main function is however not possible because of the existence of the entry point, and a jump in such a place would not be really interesting either. This being said, the call before main function issue is easy to solve: we take the old argument of the call and find the real address by subtracting the argument from 0; this operation will give us the shift needed. The same approach deals with calls after main. The jump is dealt with in the same manner, with the difference that no patch injection is needed. We did not address this issue since we established its feasibility and we already proved the feasibility of our approach with "normal" calls. Such an issue nonetheless needs to be addressed in a production system.

5.5 Shared libraries

Shared libraries are not protected in our system. They are already compiled and not included into the binary. Furthermore, we did not implement in our patch any detection mechanism for shared libraries. It could be argued that shared libraries are safer and they can therefore be left alone, but a system is as weak as its weakest link so ignoring libraries does not seem like a good idea. We therefore believe that our system should be extended to shared libraries. It should be noted that a shared library implies that two or more programs will access the same memory space, so patching the library should be approached with caution (since one program might want to patch a piece of code that is already patched). Ideally, we believe that shared libraries should be patched at the moment they are loaded by the system that is, when they are copied into the shared memory space.

5.6 Complex buffer overflow

Our work is only focused on "easy" buffer overflow. In the real world however viruses and other malicious programs are now most of the time packed or embed a cryptographic module to hide their signature. Even more deviously, they can hide their source code in order to defeat behavioral studies by anti virus software [6, 11]. Packers can be avoided easily by unpacking the binary with the execution of its first instruction, but matters become more complicated for the encrypted module. Considering this is not in the scope of our work; we believe that this is more pertinent in a discussion on kernel policy on user space program memory access. That is, countermeasures on this matter should probably be implemented deeper into the Linux kernel.

5.7 Expansion of the text segment

The expansion of the text segment for the injection of the code is not implemented. The easiest way to expand the text segment is to do so well before the code is injected, namely at compilation time. This way the binary will not corrupted and the text segment will have the appropriate size to accommodate the patch. The goal of our patch however is not to modify the compilation process, so such an easy approach is not acceptable. The allocation of the text segment is deeper in the kernel compared with the place of our modifications, but modifying it should be feasible. Given that this functionality is critical to our system we regard this shortcoming as the major flaw of our approach.

We are able to expand the text segment at the level of our work but this does pose a number of issues. The data segment start just at the end of the text segment, and the text segment cannot grow in the opposite direction of the data segment. Indeed there is nothing before the text segment, and so the addresses will no longer be valid. Then the text segment should grow toward the data segment, meaning that the data segment itself should be shifted, eating into the heap space. The main complication of such an approach is that it implies the reallocation and modification of all statically accessed data.

5.8 Patch on all instructions

Our proof of concept patches just three instructions: a ret, a call, and a simple jump. A CISC processor becomes a problem not only because of the variable-length instruction set (that necessitated the use of a disassembler with all the negative effects outlined above) but also because several more opcodes accomplish the same thing and so should be patched as well. Unfortunately most computers have such an instruction set, so no matter how hindering this architecture is, the whole bunch of call, jump, and ret-equivalent instructions should be patched in a production system.

5.9 Dependence of the *probes module

The usage of probe family and more generally the usage of module allows a fast development, modularity, and an amazing interface. The probe family gave us the power to perform temporary modifications in a lot of different places without kernel recompilation. However, probe family is better used for testing (as a temporary solution) rather than final implementations. Indeed the working of probes is based on hooking functions, but these functions can disappear with a new kernel version. In addition, the address of a function might change with every kernel recompilation, so probes should be reconfigured every time the kernel is compiled.

5.10 Optimization, crashes and kernel modification

Modules cannot use all the kernel functions, but only those functions that are exported. For this purpose the function access_remote_vm had to be exported. The implications of such an export need to be investigated.

Optimization was not considered at all at this stage of our work. There are a number of good candidates for such. Most notably the use of disassembler and Python scripts are sub-optimal. Using external tools works fine for a proof-of-concept such as ours, but optimization should be given a hard look for a production system.

Some freezes even crashes on big applications exist and are caused by two issues. First, the time it takes for the patching process on a large piece of code will often exceed various kernel timeouts (especially from the scheduling system). Secondly, we did not patch all the versions of the call and ret opcodes (as detailed above, the number is large for a CISC processor); the mix of patched and unpatched opcodes could thus lead to incorrect behaviour. This shortcoming is however the easiest to fix.

Scheduling was a problem during the development of the patch. Indeed, we delay the execution of the program, but in the kernel a program has a spot into the execution queue and if this execution is delayed, the schedule can crash. This issue needs careful consideration, but in the process management rather than memory management system. An elegant solution would be to pause the program at the beginning of the patching process and resuming it afterward. This should be easy to implement, as the mechanisms for suspending and resuming a program are already present.

Note that none of these problems are show-stoppers. Fixing them is no longer a matter of “how” but more a matter of spending time to code their solutions. We therefore regard out system in its present proof-of-concept form as a substantial step forward toward more secure computing systems.

6 Conclusions

At the beginning of our work we expected good theoretical and practical solutions for stack overflow avoidance. In particular we expected to be able to block a program if an exploitation is tested on that program. This response should be available at the user level, after the conception and the compilation of the program. This will largely compensate for negligent development and protect the users of legacy software.

We started this project with the feeling that developers have some methods to counter buffer overflows at their disposal, including code checking, static analysis, debugging, etc. However the end-user often does not understand the problem and may not even be aware that a problem exists in the first place. The existing end-user protection either requires the intervention of third-party software (or even hardware) or is bypassable (NX-bit, ASLR). Our solution works on the user side, without compilation and without third party components. It can be implemented in the standard kernel of the operating system.

Our focus has been the simple buffer overflows (on the stack), which are pretty easy to discover and exploit. Most of the time they are not dangerous for the system, but in a non-negligible amount of cases they are really dangerous. They are the most popular buffer overflow and therefore a considerable problem.

Throughout our work we had in mind the following two goals:

- *Just a few operations added:* The biggest potential problem for this kind of patch is that adding operations for each call and return of a function in the code can easily incur a severe performance penalty. It is therefore really important to create a light patch, with just a few operation added. Indeed, the function is one of the basic entity in the C language and it is therefore used all the time. Adding too much code to the patch will result in severely degraded performance, which could be especially bad for embedded systems.

- *Backward compatible:* The idea of the patch is to be effective for all the software, without the need of any compile-time protection measures or source modifications. The patch should be effective in particular for legacy software. The only problem the patch will not be able to address is its absence from a particular kernel on a particular machine.

We believe that we have been fully successful at a theoretical level and that we have also provided a good starting point for a practical application. We have violated to some degree our first goal (minimal overhead), but it should be noted that the bulk of the overhead introduced by our system happens at the beginning of the execution of the program; the overhead is minimal once the actual execution has started.

We believe that our work has the potential of simplifying the area of computer security considerably. We did not produce a production-level system but we explored a significant IT security issue and we have effectively shown that guarding against buffer overflows at run time is not only possible but also feasible.

Some old worms and other virus could still run unaffected by my patch. However, it is unlikely that programs are not patched against them. Some new threats may use a completely different approach, that side-steps our system. Today, the kernel defense against malware is based on several separate technologies including ASLR, NX-BIT, and stack canaries. We believe that trying to merge all the defenses into one big subsystem would be a big mistake. Indeed, some protections seem redundant but in certain situations they have their own usefulness. Our patch is not here to replace the old protections but to fill the gap opened by their weaknesses and add a new protection.

6.1 Kernel wish list

The following discussion is more philosophical in nature than the one in the previous section. While working on our system we have stumbled upon several shortcomings of the current Linux kernel. We will therefore try to address now the more general issue of what functionality we would like to emerge in the Linux kernel in an ideal world. Our wish list is not directly related to the buffer overflow topic.

Our first wish is, of course, a powerful kernel debugging interface, for both the kernel itself and for the programs being run under its control. After all, the kernel own all the rights in the system, so it is one of the best places to follow the behaviour of all the programs. Kgdb exists but cannot be use on modules and also needs a second computer. We would like to see something between objdump, hexdump, ollydbg, ida and gdb. As an example, we cannot ask objdump to send us just the assembly of a specified function. There is to the best of our knowledge no powerful interface to inject code into an elf file and even less to inject it directly into a process. However, such a thing is feasible with access_remote_vm and could be really useful. We saw in this manuscript how such an injection is useful for preventing buffer overflows, but other uses can be easily envisioned. For instance such a mechanism will facilitate statistics on canaries (stack or otherwise) and maybe discover a lack of randomness or some other issue.

Our second wish is related to the kernel source code. The code itself is proper and well organized but some files do not include any comments. Documentation does exist, but sometimes it is insufficient to understand a small part of a function and some other time is lacking. At the same time, the name of variables created in the 90's are quite tough to understand and should ideally be brought to the modern naming scheme. True, the Linux Kernel is an open-source project so we can find and investigate the source code, but we also need documentation (inside as well as outside the kernel) for a proper understanding. We were faced quite often with the thankless task of figuring out why a function was written like that or why it returned this value, or worst, what is the acronym of this variable and what it means. Currently the only way to answer these questions is to follow the functions as they are called, sometimes through two or three different levels.

As stated earlier the easiest way to manipulate the memory of a program (expansion, injection, deleting, reduction of text segment, bss, data, etc.) is by a function inside the kernel. Some functionality in this respect exists, but we believe that expanding this capability is a worthy pursuit. The fixed aspect of a binary in memory is needed for safety and stability, but easy modification of binaries in memory is useful for developers. A system such as ours in particular would benefit greatly from such a support.

References

[1] J. P. ANDERSON, *Computer security technology planning study*, 1972. csrc.nist.gov/publications/history/ande72.pdf, p. 61.

[2] *Address space layout randomization*, Mar. 2003. pax.grsecurity.net/docs/aslr.txt (retrieved Nov. 2012).

[3] Bulba and Kil3r, *Bypassing stackguard and stackshield*, Phrack, 10 (2000). phrack.org/issues.html?issue=56&id=5.

[4] c0ntex, *Bypassing non-executable-stack during exploitation using return-to-libc*. www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf (retrieved Nov. 2012).

[5] CERT/CC, *Advisory CA-2001-19 "Code Red" worm exploiting buffer overflow in IIS indexing service DLL*, Jan. 2002. www.cert.org/advisories/CA-2001-19.html (retrieved Sep. 2013).

[6] Delikon, *Changing the pe-file entry-point to avoid anti-virus detection*, Mar. 2004. repo.zenk-security.com/Reversing%20.%20cracking/EN-Changing%20the%20entry-point.pdf.

[7] H. Etoh, *GCC stack-smashing protector (for gcc-2.95.3)*. "gcc-patches" mailing list, June 2001. gcc.gnu.org/ml/gcc-patches/2001-06/msg01753.html (retrieved Nov. 2012).

[8] *Exploit-db*. www.exploit-db.com.

[9] D. Goodin, *Puzzle box: The quest to crack the world's most mysterious malware warhead*. Arstechnica, Mar. 2013. arstechnica.com/security/2013/03/the-worlds-most-mysterious-potentially-destructive-malware-is-not-stuxnet.

[10] *Hackerzvoice*. www.hackerzvoice.net.

[11] *Intel 64 and ia-32 architectures software developer's manual combined volumes 2a, 2b, and 2c: Instruction set reference, a-z*. download.intel.com/products/processor/manual/325383.pdf (retrieved Jul. 2013).

[12] B. Kerouanton, *Reinventing old school vulnerabilities*, Apr. 2012. www.youtube.com/watch?v=5KK-FT8JLFw (retrieved Nov. 2012).

[13] D. Knowles, *W32.SQLExp.Worm*, Feb. 2007. www.symantec.com/security_response/writeup.jsp?docid=2003-012502-3306-99 (retrieved Sep. 2013).

[14] E. Levy (aka Aleph One), *Smashing the stack for fun and profit*, Phrack, (1996).

[15] *Non-executable pages design and implementation*, May 2003. pax.grsecurity.net/docs/noexec.txt (retrieved Nov. 2012).

[16] *Nuit du hack*. www.nuitduhack.com.

[17] B. M. Padmanabhuni and H. B. K. Tan, *Defending against buffer overflow vulnerabilities*, Computer, 44 (2011), pp. 53–60.

[18] P. Panchamukhi, *Kernel debugging with kprobes*. IBM DevelopersWorks, Aug. 2004. www.ibm.com/developerworks/library/l-kprobes/index.html.

[19] *Phrack*. www.phrack.com.

[20] P. Rascagneres, *Voyage au centre du SSP–Linux*, Feb. 2012. www.r00ted.com/doku.php?id=voyage_au_centre_du_ssp_linux (retrieved Nov. 2012).

[21] D. Seeley, *A tour of the worm*. web.archive.org/web/20070520233435/http://world.std.com/~franl/worm.html (retrieved Nov. 2012).

[22] H. Shacham, M. Page, E.-J. B. Pfaff, Goh, N. Modadugu, and D. Boneh, *On the effectiveness of address-space randomization*, in Proceedings of the 11th ACM Conference on Computer and Communications Security, 2004, pp. 298–307.

[23] Z. Shao, J. Cao, K. C. C. Chan, C. Xue, and E. H.-M.Sha, *Hardware/software optimization for array & pointer boundary checking against buffer overflow attacks*, Journal of Parallel and Distributed Computing, 66 (2006), pp. 1129–1136.

[24] Z. Shao, C. Xue, Q. Zhuge, and E. H.-M. Sha, *Security protection and checking in embedded system integration against buffer overflow attacks*, in Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC 2004), vol. I, Apr. 2004, pp. 409–413.

[25] TIOBE Software, *Tiobe index*. www.tiobe.com/index.php/content/paperinfo/tpci/index.html (retrieved Nov. 2012).

[26] A. Vandecappelle, *Kernel memory allocation*. Linux Kernel Newbies, Feb. 2008. kernelnewbies.org/KernelMemoryAllocation.

[27] *Zenk-security*. zenk-security.com.

A Binary Dissasembly scripts

- Python script to disassemble a binary

```
import os
import sys
import string

#we take args
src = ’/tmp/’
#src = ’/home/user/poc’
path1 = sys.argv[1]
path2 = sys.argv[1]
path2 = path2.replace(’/’, ’.’)

#we call objdump
a = os.popen(’/usr/bin/objdump -j.text -d ’ + path1).read()

#we send the address of the begining of the main and
b = a.split(’<frame_dummy>:’)
b = b[1].split(’\n\n’)
address_first = b[1].split(’ ’)

#we take the address of the first instruction of the text segment not shared
a = a.split(address_first[0])
a = a[1].split(’<__lib’)

a = a[0].split(’\n’)
path2 = ’1’
#loop to catch call jump and ret
f = open(src + path2, ’w’)
for b in a:
    if (b.find("jmp") > 0):
        b = b.split(’\t’)
        f.write(’0’ + b[0].split(’:’)[0].split(’ ’)[1] + " " + ’2\n’)
        b[2].split(’ ’)[0]
    elif(b.find("call") > 0):
        b = b.split(’\t’)
        #we take the address pointed by the call
        print a[len(a) - 1]
        print ’0’+b[2].split(’ ’)[3]
        print address_first[0]
        print"\n"
        if(((’0’+b[2].split(’ ’)[3]) < (a[len(a) - 1])) & ((’0’+b[2].split(’ ’)[3])
             > (address_first[0]))):
            f.write(’0’ + b[0].split(’:’)[0].split(’ ’)[1] + " " + ’0\n’)
        b[2].split(’ ’)[0]
    elif(b.find("ret") > 0):
        b = b.split(’\t’)
        f.write(’0’ + b[0].split(’:’)[0].split(’ ’)[1] + " " + ’1\n’)
        b[2].split(’ ’)[0]
f.close()
f2 = open(src + path2 + ’2’, ’w’)
f2.write(address_first[0] + ’\n’ + b)
```

```
f2.close()
```

- C wrapper for the Python script, to be called from a kernel module

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char *truc = "/home/user/wrappertest/a.out";
    char *cmd = "python /home/user/wrappertest/dat.py ";
    char *final_cmd = malloc((strlen(cmd) + strlen(argv[1] + 1) * sizeof(char)));
    strncpy(final_cmd, cmd, strlen(cmd));
    strncpy(final_cmd + strlen(cmd), argv[1], strlen(argv[1]));
    final_cmd[strlen(cmd) + strlen(argv[1])] = ’\0’;
    system(final_cmd);
}
```

B The my_call.c module

```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/kprobes.h>
#include <linux/sched.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/mm_types.h>
#include <linux/ptrace.h>
#include <linux/list.h>
#include <linux/slab.h>


void stacking_eip(void)
{
    //variables
    struct mm_struct *mm;
    struct Alt_stack *insert;
    struct pt_regs *my_regs;

    //allocation
    mm = current->mm;
    insert = kmalloc(sizeof(*insert), GFP_KERNEL);
    my_regs = task_pt_regs(current);

    //we pick up eip
    //the current eip is on the int 0x80
    //we have to add size of call to the current eip
    //sizeof(call) = 5
    insert->eip = my_regs->ip + 7;
    //we create the node
    INIT_LIST_HEAD(&insert->mylist);

    //we add it at the end of the linked list
    list_add_tail(&insert->mylist, &mm->alt_eip_stack.mylist);
}

int Pre_Handler(struct kprobe *p, struct pt_regs *regs)//, unsigned long flags)
{
    //we hook just the kernel space
    if(current_uid() != 0)
    {
        stacking_eip();
        printk("CALL\n");
    }
    return(0);
}

void Post_Handler(struct kprobe *p, struct pt_regs *regs, unsigned long flags)
{
}

static struct kprobe kp;

int myinit(void)
{
    printk("module inserted\n");
    kp.pre_handler = Pre_Handler;
    kp.post_handler = Post_Handler;
    kp.addr = (kprobe_opcode_t *)0xc1045cac;
    register_kprobe(&kp);
    return(0);
}

void myexit(void)
{
    unregister_kprobe(&kp);
    printk("module removed\n");
}

module_init(myinit);
module_exit(myexit);
MODULE_AUTHOR("BEN");
MODULE_DESCRIPTION("my_call hook");
MODULE_LICENSE("GPL");
```

C The my_ret.c module

```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/init.h>
```

```
#include <linux/kprobes.h>
#include <linux/sched.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/ptrace.h>
#include <linux/list.h>
#include <linux/slab.h>
#include <linux/mm_types.h>
#include <linux/list.h>
#include <linux/sys.h>


void unstacking_eip(void)
{
    //variables
    struct mm_struct *mm;
    struct Alt_stack *remove, *tmp;
    struct pt_regs *my_regs;
    unsigned long eip = 0;
    unsigned char temp[4];

    //allocation
    mm = current->mm;
    my_regs = task_pt_regs(current);

    //we take the value of eip stacked
    //it should be the top of the stack
    access_remote_vm(current->mm, my_regs->sp, temp, 4, 0);
    eip += temp[0];
    eip += temp[1] * 0x100;
    eip += temp[2] * 0x10000;
    eip += temp[3] * 0x1000000;

    //we check if eip stacked is the same than the value stored into the linked list
    //we loop the linked list, the last element will be the last stacked !
    list_for_each_entry(remove, &(mm->alt_eip_stack.mylist), mylist)
    {
        //we do nothing, indeed we just want to travel the linked list
        tmp = remove;
    }
    //here the structure remove got the last element of the linked list
    //and we compare his eip value with the value on the stack
    if((tmp->eip - eip - 0x2) != 0x0)
    {
        //if it's not the same value, we stop the program and delete the entire
        //linked list
        //SIGKILL is sended because the program is consided as corrupted and
        //a SIGQUIT give the control back to the program and it should be avoided
        //so => SIGKILL
        printk("STOP THE SOFT !\n");

        //and we clean the alt eip
        list_for_each_entry_safe(remove, tmp, &(mm->alt_eip_stack.mylist), mylist)
        {
            list_def(remove->mylist);
            kfree(remove);
        }
        kill(current->pid, 9);
    }
    else
    {
        //else, the program is safe, we remove the last element (struct remove)
        //and let the program running
        list_del(&tmp->mylist);
        kfree(tmp);
        printk("START THE SOFT !\n");
    }
}

void Pre_Handler(struct kprobe *p, struct pt_regs *regs, unsigned long flags)
{
}

void Post_Handler(struct kprobe *p, struct pt_regs *regs, unsigned long flags)
{
    if(current_uid() != 0)
    {
        unstacking_eip();
    }
}

static struct kprobe kp;

int myinit(void)
{
    printk("module inserted\n");
    kp.pre_handler = Pre_Handler;
    kp.post_handler = Post_Handler;
    kp.addr = (kprobe_opcode_t *)0xc1045cc0;
    register_kprobe(&kp);
    return(0);
}

void myexit(void)
{
    unregister_kprobe(&kp);
    printk("module removed\n");
}

module_init(myinit);
module_exit(myexit);
MODULE_AUTHOR("BEN");
MODULE_DESCRIPTION("KPROBE TEST");
MODULE_LICENSE("GPL");
```

D The do_exec.c module

```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/kprobes.h>
#include <linux/sched.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/ptrace.h>
#include <linux/list.h>
#include <linux/slab.h>
#include <linux/mm_types.h>
#include <linux/kmod.h>
#include <linux/string.h>
#include <linux/buffer_head.h>
#include <linux/delay.h>
struct file* file_open(const char *path, int flags, int mode)
{
    struct file* filp = NULL;
    mm_segment_t oldfs;
    int err = 0;

    oldfs = get_fs();
    set_fs(get_ds());
    filp = filp_open(path, flags, mode);
    set_fs(oldfs);

    if(IS_ERR(filp))
    {
        err = PTR_ERR(filp);
        return(NULL);
    }
    return(filp);
}

void file_close(struct file* the_file)
{
    filp_close(the_file, NULL);
}

int file_read(struct file* file, unsigned long long offset,
              unsigned char* data, unsigned int size)
{
    mm_segment_t old_fs;
    int ret = 0;

    old_fs = get_fs();
    set_fs(get_ds());

    ret = vfs_read(file, data, size, &offset);

    set_fs(old_fs);
    return ret;
}


static int uhm(void)
{
    char *argv[] = {"/home/user/wrappertest/a.out\0", "/home/user/poc/vuln4\0", NULL};
    static  char *envp[] =
    {
        "HOME=/",
        "TERM=linux",
        "PATH=/sbin:/bin:/usr/sbin:/usr/bin",
        NULL
    };

    return call_usermodehelper_fns(argv[0], argv, envp, 1, NULL, NULL, NULL);
}


void create_list_node(void)
{
    //we initialize the linked list for this task
    struct mm_struct *mm = current->mm;
    INIT_LIST_HEAD(&mm->alt_eip_stack.mylist);
}

//unsigned int *disass[2] is an 2D array, first cell is the address
//second cell is the type (0 = call, 1 = ret, 2 = jmp)
int read_memory_process(char *path1, char *path2)
{
    //variables
    unsigned char buf[1];
    unsigned int i = current->mm->end_code;
    unsigned char patch_call[7] = {0xb8, 0x5f, 0x01, 0x00, 0x00, 0xcd, 0x80};
    unsigned char patch_ret[7] = {0xb8, 0x60, 0x01, 0x00, 0x00, 0xcd, 0x80};
    unsigned int shift = 0;
    unsigned int j = 0, k = 0;
    unsigned int disass[90][2];
    unsigned char tmp_read[180];
    unsigned char tmp_add[8];
    unsigned int nb_read = 0;
       unsigned int nb_elem = 0;
       unsigned int real_start;
       unsigned int real_end;
    struct file *f = NULL;
    unsigned int cmp = 0;
    unsigned char add_read[4];
    unsigned char temp_int_add = 0;
    unsigned int temp_shift = 0;

    //we read the second document
    f = file_open(path2, O_RDONLY, 0);
    if (f == NULL)
    {
```

```
    //ERROR
    printk("FIRST\n");
    return(-1);
}
//we read the entire
nb_read = file_read(f, 0, tmp_read, 20);
//we copy the first addresse and the second
strncpy(tmp_add, tmp_read, 8);
sscanf(tmp_add, "%08x", &real_start);
strncpy(tmp_add, tmp_read + 9, 8);
sscanf(tmp_add, "%08x", &real_end);

file_close(f);
f = NULL;

//we read the first document
f = file_open(path1, O_RDONLY, 0);
if (f == NULL)
{
    //ERROR
    printk("SECOND");
    return(-1);
}
//we read the entire file
nb_read = file_read(f, 0, tmp_read, 2000);

//we loop to put everything at the good place
while(j < nb_read)
{
    //we read the addresse 8 bits
    strncpy(tmp_add, tmp_read + j, 8);
    sscanf(tmp_add, "%08x", &disass[k][0]);
    //we shift and drop the space
    j += 9;
    // x - 48 value in ascii to decimal
    disass[k][1] = tmp_read[j] - 48;

    //we shift to align to the first char of the next line
    j += 2;

    //we inc the nb_elem
    nb_elem++;
    cmp++;
    k++;
}
//we start from 0 each time so we dec the cmp
cmp--;

file_close(f);

j = 0;
k = 0;

//we calculate the shift
```

```
for (j = 0 ; j < nb_elem ; j++)
{
    //if the type != 2 we have a shift
    if (disass[j][1] != 2)
    {
        shift += 7;
    }
}

//we start to copy from the end_code - shift, the add of nop gave us space
i = real_end - shift - 17;

while(i > real_start)
{
    if(cmp + 1 > 0 && i <= disass[cmp][0])
    {
        //we have something to do and the instruction is already wrote
        if(disass[cmp][1] == 0)
        {
            //call part
            //is the call point to the text segment ?
            //if not, no patch
            if((disass[cmp][0] > real_start) && (disass[cmp][0] < real_end))
            {
                //we change the add
                //we copy the prefix of the call first
                access_remote_vm(current->mm, i, buf, 1, 0);
                access_remote_vm(current->mm, i + shift , buf, 1, 1);

                //we read args of the call
                access_remote_vm(current->mm, i + 1, add_read, 4, 0);
                temp_int_add = 0x000000 + add_read[0];
                //we check if the call is in the range of the real text segment
                //if not we do nothing, the rest of the patch will copy the old
                //value
                if(((i + shift + temp_int_add + 5) >= real_start) &&
                   ((i + shift + temp_int_add + 5) <= real_end))
                {
                    temp_shift = 1; //we start from one to do not forget
                                    //the actual injection
                    //we loop the array to find the injection BETWEEN the call
                    //and the function called
                    for (j = cmp ; disass[j][0] > i && j >= 0 ; j--)
                    {
                        if ((disass[j][0] < (disass[cmp][0] + temp_int_add + 5)))
                          //&& (disass[j][0] < (disass[cmp][0] - temp_int_add +5)))
                        {
                            if(disass[j][1] != 2)
                            {
                                temp_shift++;
                            }
                        }
                    }
                    //now we recalculate the shift
                    add_read[0] = temp_int_add + (temp_shift * 7);
                    //and we rewrite the call
                    access_remote_vm(current->mm, i + 1 + shift, add_read, 4, 1);
                    if ((shift - 6) >= 0)    /* // */
                    {
                        shift -= 7;
                    }
                    //we inject the patch
                    access_remote_vm(current->mm, i + shift, patch_call, 7, 1);
                    i--;

                }
            }
        }
        else if(disass[cmp][1] == 1)
        {
            //ret part
            //first we copy the ret op code
            access_remote_vm(current->mm, i, buf, 1, 0);
            access_remote_vm(current->mm, i + shift, buf, 1, 1);
            //we inject the patch
            if ((shift - 6) >= 0)    /* // */
            {
                shift -= 7;
            }
            access_remote_vm(current->mm, i + shift, patch_ret, 7, 1);
            i--;
        }
        else if(disass[cmp][1] == 2)
        {
            //jmp part
            //is the jump point to the text segment ?
            //if not, no patch
            if((disass[cmp][0] > real_start) && (disass[cmp][0] < real_end))
            {
                //we change the add

            }
        }
        else
        {
            //ERROR !
        }
        cmp--;
    }
    else
    {
        //we copy the rest of the program if no modification is needed
        access_remote_vm(current->mm, i, buf, 1, 0);
        access_remote_vm(current->mm, i + shift, buf, 1, 1);
        i--;
    }
}
access_remote_vm(current->mm, 0x0804823e, buf, 1, 0);
```

```
    return(0);
}


int Pre_Handler(struct kprobe *p, struct pt_regs *regs)
{
    //do nothing
    return(0);
}

void Post_Handler(struct kprobe *p, struct pt_regs *regs, unsigned long flags)
{
    if(current_uid() != 0)
    {
        struct mm_struct *mm = get_task_mm(pid_task(find_get_pid(current->pid),
                                                                PIDTYPE_PID));
        down_read(&mm->mmap_sem);
        uhm();
        if(read_memory_process("/tmp/1", "/tmp/12") == -1)
        {
            printk("ERROR at the file reading.\n");
        }
        create_list_node();
        up_read(&mm->mmap_sem);
    }
}

static struct kprobe kp;

int myinit(void)
{
    printk("module inserted\n");
    kp.pre_handler = Pre_Handler;
    kp.post_handler = Post_Handler;
    kp.addr = (kprobe_opcode_t *)0xc1001596;
    register_kprobe(&kp);
    return(0);
}

void myexit(void)
{
    unregister_kprobe(&kp);
    printk("module removed\n");
}

module_init(myinit);
module_exit(myexit);
MODULE_AUTHOR("BEN");
MODULE_DESCRIPTION("KPROBE TEST");
MODULE_LICENSE("GPL");
```

E Makefile for modules

```
obj-m +=my_module.o
KDIR= /lib/modules/$(shell uname -r)/build
all:
        $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
clean:
        rm -rf *.o *.ko *.mod .c* .t*
```

F Simple test program

```
void c();
int main(void)
{
    c();
}
void c(void)
{
}
__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");
__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");
__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");
__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");
__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");
__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");
__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");
```

G Simple test program already patched (to test system call)

```
void c();
int main(void)
{
    __asm__("MOV $351, %eax");
    __asm__("INT $0x80");
    c();
}
void c(void)
{
    __asm__("MOV $352, %eax");
    __asm__("INT $0x80");
}
__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");
__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");
__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");
__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");
__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");
__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");
__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");
```

H Program with a buffer overflow

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char sc[]= "\x10\x84\x04\x08";
void p();

void c()
{
    printf("Bufferoverflow\n");
}

int main(int argc, char *argv[])
{
    p("ab");
}

void p(char *string_to_copy)
{
    char str[8];
    for(int i = 0 ; i < 58 ; i++)
    {
        str[i] = sc[i % 4];
    }
    printf("a");
}

__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");
__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");
__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");
__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");
__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");
__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");
__asm__("NOP");__asm__("NOP");__asm__("NOP");__asm__("NOP");
```

I Compilation command for testing programs

This command line will remove all the protections from the compilation process:

```
gcc source.c -w -O0 -ggdb -std=c99 -static -D\_FORTIFY\_SOURCE=0 \
  -fno-pie -Wno-format -Wno-format-security -fno-stack-protector \
  -z norelro -z execstack -o output.out
```

J Previous work

We present here our alternate source of research that is non-academic in nature. It consists in blogs of independent IT security consultants and researchers, repositories of community knowledge, and sometimes directly the community through conventions and other meetings:

- *Phrack* [19] (repo of webzine): Phrack is a webzine where we can find some papers on computer science and particularly IT security, such as the famous "Smashing the stack for fun and profit" [13], This resource contains old but also new, original papers.

- *Exploit-db.com* [8] (repo): Exploit-db is a repository similar to the dead milw00rm where we can find numerous security vulnerabilities and proof-of-concepts for their exploitation. This resource is really useful to know if a particular program (or a particular version) is vulnerable.

- *Zenk-security* [27] (repo and community): Zenk-security is a French community centered on IT security. Without a professional structure, it is more a passionate group where we can find some help from personalities of the French IT security.

- *Hackerzvoice* [10] (repo and community): Hackerzvoice is a French community like Zenk-security, with quite the same people but featuring public access.

- *Nuit Du Hack* [16] (convention): The Nuit Du Hack is a French convention on IT security like DefCon but smaller. Some of people from Zenk-security and Hackerzvoice talk on security problem and issues.

All of those non-academic resources are interesting because of their more technical aspect, closer to the IT security industry. Indeed, communities often take less time than research facilities to speak about a problem and are more in the present. For example the result of the "0 day java" query to a search engine will consist largely in blogs wrote by member of the IT security community. When the community is more "cutting edge" we can have access to resources even before their official announcement. Overall through communities we can be closer to the present.