Toward Preventing Stack Overflow Using Kernel Properties

Benjamin Teissier and Stefan D. Bruda
Department of Computer Science,Bishop's University, 2600 College St, Sherbrooke, Quebec J1M 1Z7, Canada

Keywords: Software and Information Security, Buffer Overflow, Stack Overflow, Privilege Escalation, Kernel, Linux.

Abstract: We contribute to the investigation of buffer overflows by finding a more accurate way of preventing their exploitation. We work 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. The kernel injects at launch time some (minimal) code into the binary being run, and subsequently uses 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. In addition we take the first steps toward implementing such a defense.

1 INTRODUCTION

IT security is a complex and important field with 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 security threats in general and privilege escalation in particular deserve a greater attention.

One of the oldest issues in this field is buffer overflow. Its range of exploitations in particular is very large, ranging from data leaks to taking over a complete computing system. These are all dangerous: 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: buffer overflow is first mentioned as early as 1972 (Anderson, 1972), and the first documentation of a hostile exploitation was written in 1988 (Seeley, 2007). However, as can be seen during events such as the NDH 2k11 "old hacking" conference (Kerouanton, 2012), 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 gaining access to the system, including "SQL Slammer" (Knowles, 2007) and "Code Red" (CERT/CC, 2002). They infected a substantial number of computers and servers, compromising data, consuming computing time and compromising the security of the entire system.

In its early history the exploitation of buffer overflows was reserved for the elite of hackers. This ceased to the case with the publication of the famous Phrack paper "Smashing the stack for fun and profit" (Levy, 1996). The paper popularized buffer overflow but also provided a good and complete "how to". All of a sudden buffer overflow exploitation became accessible to everybody.

In all, buffer overflow is an old but at the same time a current and acute problem. One can see this just by looking at the exploit-db.com Web site, an archive of exploits and vulnerable software. More than 100 pages therein (with some 20 articles per page) are dedicated to buffer overflow vulnerabilities.

Older program running on on older computers often cannot be patched for buffer overflow bugs. The reasons range from lack of expertise to source code unavailability. This motivates our goal, which is to investigate means of stopping the exploitation of stack overflows (a particular kind of buffer overflow) for all the programs running on a particular machine instead of cleaning overflow conditions from all the pieces of code one by one. Our thesis is 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. This in turn addresses the problem automatically for all the programs running on a machine.

To substantiate this thesis we propose a patch to the Linux kernel that mitigates the stack overflow issue. We work on the Linux kernel since it is open source and is also modifiable by the computing community at large (so it will be possible to have our patch accepted in the official kernel source once a production variant is reached).

Our patch is a proof of concept that needs substantial further refinement to become a production system. Such a refinement is however no longer a matter of "how" but more a matter of spending time to write the necessary code; a roadmap toward an actual production system is described in Section 5. We therefore regard out system in its present proof-of-concept form as a substantial step forward toward more secure computing systems.

2 BUFFER OVERFLOW

Buffer overflow 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 the corruption of another part of the program's memory. This results in unpredictable behavior, ranging from erroneous results to system crashes. Often buffer overflows can be exploited by malicious entities.

A *stack overflow* is a buffer overflow happening on the stack. The typical cause is the overflow of an array stored on the stack. Often caused by the use of strcpy() (which does not check for the size of the destination array), the stack overflow is the most popular and the easiest buffer overflow to detect (and then exploit).

A stack overflow attack targets the EIP register and so 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. A carefully crafted usage of the overflow of a data structure placed on the stack will overwrite the EIP value pushed on the stack and so change the program flow. When the current function "returns" the altered EIP value produces a jump to some other, malicious code that runs with the same privileges as the original code.

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

3 PREVIOUS WORK

We start by presenting general techniques developed throughout the years for mitigating the buffer overflow problem. We then summarize the recent progress on the matter.

3.1 Techniques

The *NX-bit* stands for "Never execute" and is a technique that identifies two different parts of the memory (Noexec, 2003). One part contains data, which can be overwritten but cannot be executed. The other part contains instructions and cannot be overwritten (but can be executed). The NX-bit creates a distinction in the writing permissions between the memory initialized at the start of the program (instructions, locked for writing), and the memory initialized and modified on the fly (data, unlocked for writing but not executable). The malicious code can still be loaded in the memory of the program, the EIP register can still be overwritten to point to the beginning of this code, but because of the NX-bit the injected code will still not be executed since the stack is protected for execution. This protection can however be bypassed relatively easily with a return-to-libc attack (c0ntex, 2012).

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. The POSIX system call mmap() creates in-memory clones of files or devices. The *randomization of the mmap() base* prevents to some degree attacks such as return-to-libc. The addresses of the linked shared libraries will be randomized, and so the memory location of the start of libc is no longer fixed. A brute force approach (see below) is nonetheless able to overcome such a protection.

An extension of the above technique is *ASLR* or Address Space Layout Randomization, which randomizes all the memory space of a program (ASLR, 2003): the position of the stack, the heap and third-party libraries are all randomized. The virtual addresses within the randomized space will change at each start of the program, which counters attacks based on fixed structures. Automated brute force attacks as well as the insertion of NOP instructions reduce however the effectiveness of randomization. Indeed, an attacking script can test the program quickly, and if the program crashes or cannot be exploited because EIP points to a bad address or an illegal instruction (e.g., data), the script will immediately reload the program and test again and again (Shacham et al., 2004). The NOP instruction will allow a large landing place for EIP, thus simplifying the process.

A *stack canary* is a defined value that is placed on the stack just after pushing the EIP. This value is then checked before popping the EIP at the end of the function (Etoh, 2001). This method detects the modification of EIP and so prevents in principle the takeover of the program . Stack canaries can nonetheless be bypassed (Bulba and Kil3r, 2000; Padmanabhuni and Tan, 2011).

3.2 Recent Progress

One recent paper (Rascagneres, 2012) 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 protection upon re-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. Two detection tools are finally introduced, one at the level of source code and the other for binary code. The paper essentially explains why the creation of a method to prevent buffer overflows is impossible (namely, due to the different attack methods). The shortcoming of all these methods is that they need either re-compilation, or detection before compilation. Without access to source code (and permission to compile and install software), these methods cannot be applied.

Another approach to defending against buffer overflow is at the hardware as well as software level, proposing two methods that use new assembly functions to secure the system (Shao et al., 2004). The first method, “Hardware Boundary Check” offers a simple but effective protection: When a function is called, a parallel function runs and checks the value of the target address. The protection is engaged whenever this address is equal to or larger than the value of the frame pointer. 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 checks the signature before the return and thus detects any modification. This work has several 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 as well as specialized hardware.

Boundary checking is often used in buffer overflow defenses. The main problem with such a solution remains performance: programs took two to five times more to run after the introduction of boundary checks (Shao et al., 2006). A new instruction to limit the use of resources and optimize execution was then proposed (Shao et al., 2006). The optimization of a proven method is generally a good idea but this approach continues to rely on third party components.

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

Figure 1: System call example in assembly (left) and binary (right).

4 A KERNEL PATCH

The basic idea of our patch is the kernel-space control of the user-space EIP stacking. Theoretically any kind of control on the software side is possible from within a kernel so a simple and non-intrusive such a control should be possible.

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 choices in this respect: we can either forbid the overwriting of EIP, or check the value of the stacked EIP before using it. Stack canaries take the second approach. Our work tried at first to follow the first route but ends up being close to the second approach as well, so our solution is similar to stack canaries. It is however independent on the particular binary being run: our solution is implemented into the operating system, just like ASLR.

Hooking into stack manipulation can be done in several ways. Our starting idea comes from the stack canaries protection, which adds at compile time a few instructions before every call or ret instruction. Similarly we inject small pieces of code in the program being run so that we can generate an interrupt and so enter into the kernel space (where the rest of our protection resides). As opposed to stack canaries however the injection happens at run time. The simplest and most straightforward mechanism to enter kernel space is system calls. A system call is accomplished with just two instructions, one to move the system call number into EAX and one to call the (system call) interrupt 80. An example is given in Figure 1.

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. 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 (Goodin, 2013). Moreover, the CISC instruction set features variable instruction length, which effectively prevents 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 later (namely, in Section 5).

We note that memory locking and unlocking (our initial idea) is unrealistically wasteful and so should be avoided. We eliminate the need for this 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 them into a kernel variable. 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 erroneous (and potentially dangerous) behaviour.

4.1 Implementation

Figures 2, 3, and 4 summarize the description of our system. A detailed description is then provided below. The complete patch code is also available (Teissier and Bruda, 2014).

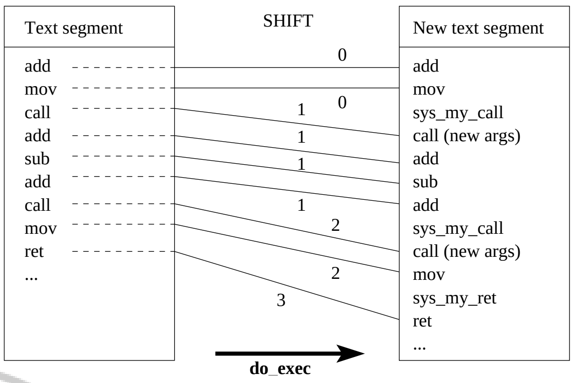

Figure 2: The do_exec module rewrites the text segment from bottom to top into a new text segment, shifting the addresses to make room for the patches (i.e., new system calls). The amount of shifting is decreased each time a patch is injected.

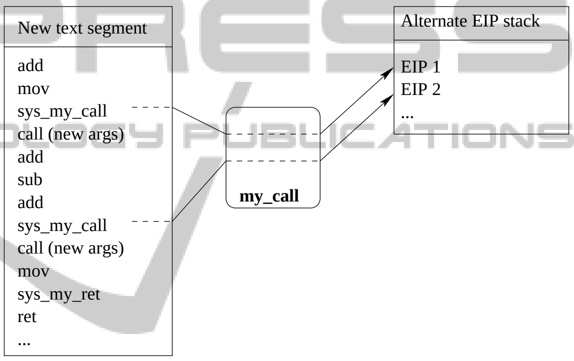

Figure 3: The my_call module probes the system call sys_my_call and copies the current EIP of the program into the alternate EIP stack.

4.1.1 Kernel Modifications

A few vital modifications into the Linux kernel were necessary:

- *mm/memory.c line 4154*

```
EXPORT_SYMBOL(access_remote_vm);
```

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

```
Struct Alt_stack alt_eip_stack;
```

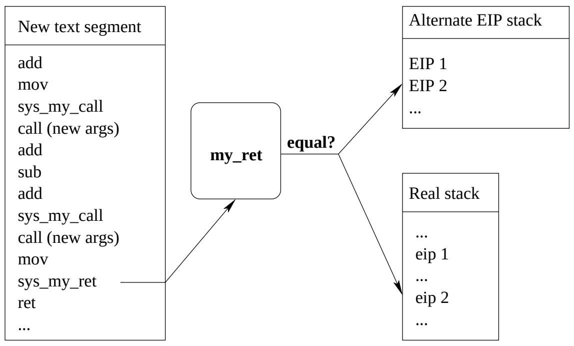

Figure 4: The my_ret module probes the systemcall sys_my_ret, check the last EIP stacked into the real stack and the last EIP stacked into the alternate stack. If they are different, the program is terminated; otherwise the alternate EIP stack is popped.

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). 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 add our custom system calls to the compilation process.

4.1.2 Module Implementation

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:

The *do_exec module* probes the kernel function "copy_process" (so the module should have been named "copy_process"; the current name stems from the original function being probed and we kept the initial 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, and 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 built 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 gives the addresses and names of the instructions of interest.

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

The *my_ret module* probes the syscall my_ret. It checks the last stacked EIP against the last EIP from the kernel linked list. The checks are performed before the ret so that potential exploitations are detected.

The modules use the *probe family, a mechanism implemented in the 2.6 version of the Linux kernel (Panchamukhi, 2004). Kprobe, Jprobe and Kretprobe

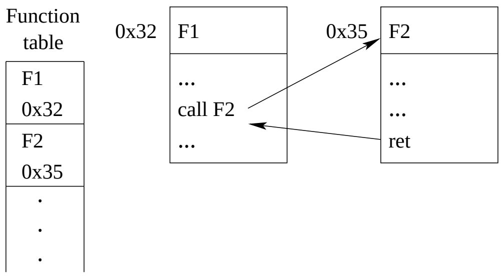

Figure 5: Normal function call: The function table includes the name of the function (top) and the address of the function in memory (bottom).

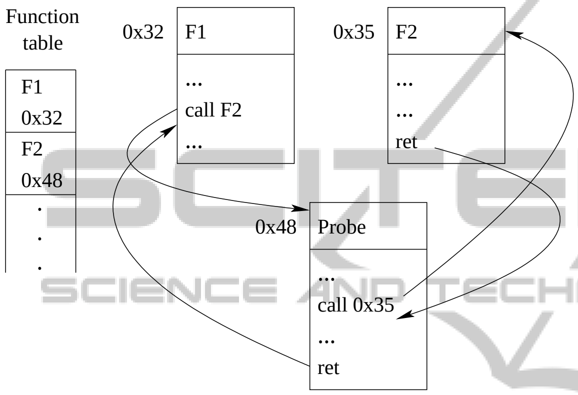

Figure 6: Probing a function: The address of F2 in the function table was changed to the address of the probe (0x48). The program flow is modified before as well as after the execution of F2.

are all members of this family. Between other things probes are accessible in modules and so eliminate the need to modify the kernel proper. The available probes are different for each mechanism. Kprobe provides one probe before and one after the target function, Jprobe provides the arguments as well as a probe before the target function, and Kretprobe provides the return value as well as a probe after the target function.

The probe family uses the hooking technique: The module using a probe for say, function "foo" will change the address used to call foo to the address of entry into that module. The module in turn handles the actual call to foo, has a chance to execute extra code both before and after this call, and eventually returns. When the probe is removed, the original address is restored. This is illustrated graphically in Figures 5 and 6.

4.2 Testing

We provided a "safe" program and an "unsafe" one. Those two programs were tested first using the techniques described in Section 3.1 (ASLR, NX-bit and stack canaries), and a second time without the ASLR protection in place but running under our system.

In the first test (that is, outside our system) the safe program executed normally and terminated properly, while the injection of malicious code in the second program worked.

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.

We argue that such a minimal testing suffices. Indeed, we effectively establish the correct handling of at least one call/return pair; however, all other such pairs are handled in the same manner by the kernel, so there is no way that their behaviour can be any different. This being said, we recognize that comprehensive testing should be performed on an eventual production system, to ensure correctness but also to validate the effectiveness of the system. Such testing is however not warranted at this stage of development (i.e., proof of concept).

5 ROADMAP TOWARD A PRODUCTION SYSTEM

The patch 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.

The modification of the text segment is functional but is not as elegant as we wanted. Indeed the use of the Python script and disassembler to find the call, ret, and jump instructions is ugly and not secure. Moreover, the whole process is slow. The CISC instruction set does not appear to allow any cleaner solution, but this issue should definitely be investigated in depth.

A program needs to know where the main function starts, but this entry point is sometimes modified by the packer. The entry point address should be rewritten if the patch modifies and moves the entry point by injecting code before its location.

The memory used by the do_exec module (which reads and modifies the binary being launched) is fixed and not optimized. Eating so much memory is detrimental, especially on the kernel side which has a limited address space (Vandecappelle, 2008). 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 any production version.

Issues related to calls before the main function are easy to solve, but we did not address them since we already proved feasibility of our approach with "normal" calls. Such an issue nonetheless needs to be addressed in a production system.

Shared libraries are not protected in our system since they are external to the binary. We believe that they should also be patched, and that the patching process should happen once, just when they are loaded by the operating system.

The expansion of the text segment for the injection of the code is not implemented. Given that this functionality is critical to our system we regard this shortcoming as the major flaw of our approach.

Our proof of concept patches just three instructions: a ret, a call, and a simple jump. One problem with the CISC instruction set is that several more opcodes accomplish the same thing and so should be patched as well.

The probe family provides an environment for fast development, modularity, and an amazing interface. However, the probe family is better used as a temporary solution rather than final implementations. Replacements should be investigated.

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.

Some freezes and 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 mix of patched and unpatched opcodes could thus lead to incorrect behaviour. This shortcoming is however the easiest to fix.

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 (Delikon, 2004; Intel, 2013). Packers can be avoided easily, but matters become more complicated for encrypted code. 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.

Note that none of the above problems are showstoppers. Fixing most of 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

Developers have at their disposal several methods to counter buffer overflows, 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 can easily escalate into 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 particularly unwelcome in 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. Section 5 provides a roadmap for future work on the matter, toward an actual production system.

Some old worms and other viruses could still run unaffected by our 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 altogether. 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 mistake. Indeed, some protections might seem redundant, but in certain situations they have their own usefulness. Our patch is not here to replace the old protection mechanisms, but rather to fill the gap opened by their weaknesses and add a new degree of protection.

REFERENCES

Anderson, J. P. (1972). Computer security technology planning study. csrc.nist.gov/publications/history/ande72.pdf, p. 61.

ASLR (2003). Address space layout randomization. pax.grsecurity.net/docs/aslr.txt (retrieved Nov. 2012).

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

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

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

Delikon (2004). Changing the per-file entry-point to avoid anti-virus detection. repo.zenk-security.com/Reversing%20.%20cracking/EN-Changing%20the%20entry-point.pdf.

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

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

Intel (2013). 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.

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

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

Levy, E. (1996). Smashing the stack for fun and profit. *Phrack*, 7(49). www.phrack.com/issues.html?issue=49&id=14.

Noexec (2003). Non-executable pages design and implementation. pax.grsecurity.net/docs/noexec.txt (retrieved Nov. 2012).

Padmanabhuni, B. M. and Tan, H. B. K. (2011). Defending against buffer overflow vulnerabilities. *Computer*, 44(11):53–60.

Panchamukhi, P. (2004). Kernel debugging with kprobes. IBM DevelopersWorks. www.ibm.com/developerworks/library/l-kprobes/index.html.

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

Seeley, D. (2007). A tour of the worm. web.archive.org/web/20070520233435/http://world.std.com/~franl/worm.html (retrieved Nov. 2012).

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

Shao, Z., Cao, J., Chan, K. C. C., Xue, C., and H.-M.Sha, E. (2006). Hardware/software optimization for array & pointer boundary checking against buffer overflow attacks. *Journal of Parallel and Distributed Computing*, 66(9):1129–1136.

Shao, Z., Xue, C., Zhuge, Q., and Sha, E. H.-M. (2004). 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)*, volume I, pages 409–413.

Teissier, B. and Bruda, S. D. (2014). An approach to stack overflow counter-measures using kernel properties. Technical Report 2014-001, Department of Computer Science, Bishop's University. cs.ubishops.ca/_media/papers/bucstr-2014-001.pdf.

Vandecappelle, A. (2008). Kernel memory allocation. Linux Kernel Newbies. kernelnewbies.org/KernelMemoryAllocation.