Instruction-level countermeasures against stack-based buffer overflow attacks

Francesco Gadaleta
Katholieke Universiteit Leuven
francesc@cs.kuleuven.be

Yves Younan
Katholieke Universiteit Leuven
yvesy@cs.kuleuven.be

Bart Jacobs
Katholieke Universiteit Leuven
bartj@cs.kuleuven.be

Wouter Joosen
Katholieke Universiteit Leuven
wouter@cs.kuleuven.be

Erik De Neve
Katholieke Universiteit Leuven
e.de.neve@gmail.com

Nils Beosier
Katholieke Universiteit Leuven
nils.beosier@beonet.be

Abstract

In this paper, we examine the possibility of using virtualization to implement a countermeasure that protects against buffer overflow attacks. The countermeasure works by adding a few extra instructions to the architecture that are emulated by the hypervisor. After running performance benchmarks, a high overhead was observed. Our proof-of-concept software implementation illustrates that the proposed approach is feasible and that the hardware implementation confirms a negligible overhead.

Categories and Subject Descriptors

H.4.m [**Information Systems**]: Miscellaneous; D.2 [**Software**]: Software Engineering

General Terms

system security

Keywords

buffer overflows, countermeasure, virtualization

1 Introduction

Virtualization has become a very popular technology. After Intel added support for virtualization to its processors [21, 31], this popularity increased even more. Windows Server 2008 comes with a hypervisor [3], which will make this technology even more widely deployed.

Since the focus for the deployment of virtualization technology has been on servers we think that a logical place to increase security seems at the level of the virtual machine monitor or hypervisor. In this paper we examine how we can use virtualization technology to better protect against buffer overflow vulnerabilities.
We show that adding this kind of countermeasure to a virtualized environment is feasible but, due to the high overhead it is not realistic.

Despite persistent research in the field, one of the most insidious vulnerabilities affecting software nowadays is still the buffer overflow. According to the NIST's National Vulnerability Database[4], 587 (10% of all reported vulnerabilities) buffer overflow vulnerabilities were reported in 2008. Almost 90% of those vulnerabilities had a high severity rating. A large amount of countermeasures have been designed to avoid buffer overflow attacks. But since many different types of attack exist and because effective countermeasures are often affected by considerable overhead, more research into this topic needs to be done.
Most of the buffer overflow vulnerabilities are located on the stack [35] and one of the most effective way to change the execution flow of the program is to modify the return address of a function. The vulnerability described in this paper normally occurs in languages that allow the programmer to access memory locations directly, without any restrictions. Thus these languages, normally used to build high performance applications, are also considered unsafe. Designing countermeasures to improve security for unsafe languages is a direction not to neglect. A survey of vulnerabilities and countermeasures for such languages is reported in [38, 18].
This paper is structured as follows: Section 2 gives a description of buffer overflow by an example. In Section 3 an overview of known countermeasures is reported together with a comparison between the two most used strategies against the vulnerability. A detailed description of our implementation in a virtualized environment is described in Section 4. An evaluation of our countermeasure is provided in Section 4.1. Section 6 presents some possible future developments and improvements. Our conclusions are provided in Section 7.

2 Problem description

A buffer overflow is the result of stuffing more data into a buffer than it can handle and may allow an attacker to control the execution flow of the attacked program.

A *return-address attack* is an attack where the attacker exploits a buffer overflow vulnerability to change the return address of a function. It is often performed together with code injection through *shellcode*. This execution of arbitrary code is what results in the high severity rating of most of the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
VTDS 2009, March 31, 2009, Nuremberg, Germany.
Copyright 2009 ACM 978-1-60558-473-7 ...$5.00

reported vulnerabilities. A typical function that is vulnerable to a buffer overflow is given in Listing 1

Listing 1: A function that is vulnerable to buffer overflow

```
char* vuln_foo(char *msg) {
   char *p;
   char buffer[30];
   p=buffer;
   strcpy(p, msg);
}
```

A standard prologue saves the frame pointer (FP) to the stack and allocates space for local variables. The epilogue restores the saved frame and stack pointer (SP) as in Listing 2.

Listing 2: The standard prologue and epilogue of vuln_foo()

```
prologue:
   pushl %ebp
   mov %esp, %ebp
   // local variables

  (vuln_foo body)

epilogue:
   leave  // copies %ebp into %esp
          // restores %ebp from stack
   ret
           // jump to address on
           // top of the stack
```

If `buffer` can be overflowed, all out-of-boundary bytes may alter the value of pointer `p`, the saved frame pointer `%ebp`, the return address, `vuln_foo()`'s arguments and so on towards higher addresses. If the return address (RET) is changed, the execution flow is hooked to the new address (if valid) and when the function returns, arbitrary code at that location will be executed.

3 Related work

Many countermeasures exist to protect against these types of problems [17, 37]. They range from safe languages [16, 23, 28, 29, 36] that remove the vulnerabilities entirely, to bounds checkers [6, 15, 25, 32] that will perform runtime checks for out-of-bounds accesses, to very lightweight countermeasures that prevent certain memory locations from being overwritten [14, 11, 39, 40] or prevent attackers from finding or executing injected code[8, 9, 10, 13, 26]. Other countermeasures will provide execution monitoring to prevent or detect applications deviating from their normal behavior [5, 27, 33]. An extensive survey of these countermeasures can be found in [37, 38]. Two countermeasures are closely related to our approach: StackShield [34] and RAD [12].

StackShield saves a copy of the return address of the function (RET) to a memory area previously allocated and restores it from this memory before returning [34]. The restored return address is not compared with the one saved in the prologue, however if detection is needed then it is trivial to add this. This strategy will keep the execution flow unchanged since the protected function always returns to its caller. Unfortunately this strategy is still affected by some drawbacks. The most relevant is that retarray is allocated in normal memory which is not the safest choice. Stackshield protects against return address attacks, not generic stack smashing. Local variables, functions' arguments, saved frame pointers may still be altered by different tricks. Some techniques to bypass this type of protections are described in [7, 30].

A higher level of security is granted by Return Address Defender (RAD) [12], however this countermeasure also comes with a higher performance penalty.

RAD automatically creates a Return Address Repository (RAR) to save return addresses. Two similar protection modes are proposed: minezone RAD and read-only RAD.

Minezone RAD sets guard pages[1] surrounding RAR (mine zones) by a mprotect() system call. This protection is executed once, at program startup. With this mechanism RAR might be altered by a direct return address modification attack [2]

(if the attacked program satisfies several special conditions [12]) without modifying the mine zones.

Read-only RAD avoid this possibility by setting the RAR read-only. This protection is executed in the prologue code of each function call, after pushing the current address into the RAR. The performance penalty introduced by this mechanism is higher than in the former since two system calls are required for each function call.

Macro and micro benchmarks show that programs protected by read-only RAD experience a slow-down ranging from 140x to 200x [12].

4 Protecting return addresses at a lower level

Although the countermeasure introduced by Read-only RAD is considered safer and harder to exploit [3] a serious performance drawback must be taken into account. The countermeasure is very expensive because of the need to do two system calls in each function's prologue.

A solution to the serious performance penalty introduced might be to implement this strategy using virtualization by adding new instructions to save and restore the return address from a read-only memory. The overhead of such a solution is expected to be much lower than the compiler version introduced by RAD.

Since virtualization is a widely deployed technology nowadays we are confident that it might represent the best scenario for such an implementation. In our implementation we emulated these new instructions in the Xen Hypervisor [2] in order to create a real-life demonstration with an extensively used virtualizing product.

[1] Guard pages are pages that have no permissions set, any attempt to access them will cause a segmentation violation

[2] A typical scenario for a direct return address modification attack is represented by the statement *A=B where A is a pointer variable and B is a variable. If there are vulnerabilities that may allow an attacker to change A and B's values, the address pointed to by the new value of A will get the new value of B. This may allow to change the content of any memory location, including return addresses.

[3] No alterations are possible on the RAR where return addresses are saved. All other areas of the stack are still not protected.

The general idea of the countermeasure described in this paper is similar to the one used by RAD. But our implementation uses special (emulated) hardware instructions to access read-only memory where all the return addresses are stored.

New code must be added to implement this feature. The assembly code generated by our modified GCC and instrumented with the new instructions is given below (Listing 3).

Listing 3: Instrumented assembly code of vuln_foo()

```
main:
   call init_callretx
   ...

vuln_foo:
 prologue:
    pushl %ebp
    mov %esp, %ebp
   // local variables
   callx
         (vuln_foo body)
 epilogue:
    retx
    leave  // copies %ebp into %esp
          // restores %ebp from stack
    ret    // jump to address on
           // top of the stack
```

A library to allocate a 4KB page and *mprotect* it has been written and called at the beginning of every program in the main function. Two new hardware instructions are used to protect the function's return address and only these two instructions can access the read-only memory previously allocated. This is achieved by adding the two instructions to the instruction set of the virtualized processor which will be trapped by the Xen hypervisor and will be emulated to save and restore the return address. In our implementation

- `callx`

  is added before the call instruction, it will save the return address onto the protected memory page

- `retx`

  is added right before the assembler `leave` instruction in function's epilogue. It will restore the return address from the protected memory page onto the stack.

Return addresses of nested functions are stored at higher addresses within the page with the aid of a counter that permits to handle return addresses in a Last-In-First-Out order. This order will be preserved until the maximum number of nested functions is reached. This number depends on the size of the *mprotected* page, which is 4KB in our implementation. Since the x86 architecture handles 32 bit addresses and a counter of the same size is required, our countermeasure can handle up to 1023 nested functions.

The basic idea used to implement this in Xen is to clear the write protection bit (WP) in the Control Register 0 (CR0) [4] before any write operation to read-only memory and then set it again. The Xen Hypervisor, which runs in supervisor mode, needs to be able to write to a read-only page from the user space memory. By unsetting the WP in CR0, the memory management unit does not check whether the page is read-only or not, allowing the new instruction to write directly.

Although Xen has the necessary code to capture illegal instructions, some setup is required to handle the new instructions' opcodes. New code that checks if the opcode we want to emulate occurred has been added. When the new instruction's opcode occurs, the user space program context (`ctxt` structure) is updated. This is required before calling `x86_emulate` which will take the context structure as parameter and performs the emulation. Before calling this function, the WP bit of `CR0` must be unset. Thus when `x86_emulate` is called, all writes to memory can happen without any fault.

New code to emulate the `callx` and `retx` instructions in the hypervisor has been added to `x86_emulate.c`.

Since we need to save the return address from current stack to memory (callx) and from memory back to the stack (retx), we need two functions that move data from one space to the other. As in a regular Linux kernel the `copy_to_user` and `copy_from_user` functions perform this task. A counter is needed to handle nested functions. This variable is incremented in `callx` and copied to the read-only memory, decremented in `retx` and copied back to the stack, to preserve a LIFO order.

A check if the return address has been altered may be performed before overwriting it with the saved value. However this will lead to a higher overhead in the overall test result.

[4] CR0 has control flags that modify the basic operation of the processor. WP bit is normally set to prevent supervisor from writing into read-only memory.

Program	Base r/t(s)	Instr. r/t(s)	Overhead
164.gzip	223	3203	13x
175.vpr	372	2892	6x
176.gcc	225	2191	8x
181.mcf	640	3849	5x
186.crafty	114	3676	31x
256.bzip2	307	5161	16x
300.twolf	717	4007	4x

Table 1: SPEC CPU2000 benchmark results of our implementation in Xen

4.1 Evaluation

To test the performance overhead we ran several integer benchmarks from the suite SPEC CPU2000 [24]. We collected results running programs instrumented with the code that implements the countermeasure and without.

All tests were run on a single machine (Intel(R) Core(TM)2 Duo CPU E6750@2.66GHz, 4096MB RAM, GNU/Linux kernel 2.6.24-xen) running Xen 3.3.0. The GCC 4.2.3 compiler has been modified to instrument assembler code with new instructions.

The benchmarks show that this implementation experiences the unacceptable factor of between 5x to 30x slow-down (Table 1). Memory overhead is 4KB, which is negligible in comparison to memory required by the program itself.

The Xen implementation is affected by a significant increase of the execution time when compared to reference

Table 2: SPEC CPU2000 benchmark results of our implementation in QEMU

Program	Base r/t(s)	Instr. r/t(s)	Overhead
164.gzip	1368	1446	1.05x
176.gcc	1010	1067	1.05x
181.mcf	646	701	1.08x
186.crafty	1542	1656	1.07x
197.parser	2652	2844	1.07x
255.vortex	2458	2606	1.06x
256.bzip2	1638	1729	1.05x
300.twolf	2316	2399	1.03x

time. That is mostly due to the context switch to the hypervisor that is needed to perform the emulation.
We conclude from this, that this type of countermeasure, while technically feasible and faster than RAD, does not have a realistic chance of deployment except in higher security environments.

5 Optimization through hardware support

The overhead introduced by Xen lead us to implement the same countermeasure in an emulated environment in order to show that this countermeasure may have a real life deployment if implemented in hardware. QEMU [1], a processor emulator, has been used for this purpose.

We added the same instructions to the emulated x86 instruction set. Although our countermeasure has been performed on an emulated x86 architecture, an implementation for all the other emulated processors is straightforward since we have created a store instruction common to all architectures emulated by QEMU [1].

A new Memory Management Unit (MMU) mode for the emulated x86 processor has been created.
When a normal store function is executed with this new mode, the MMU performs the translation from virtual to real address and automatically enables write permissions right before adding the translated entry to the Translation Lookaside Buffer (TLB)[22].

With this mechanism we allow callx and retx instructions to perform writes to the protected memory in order to save and restore the return address respectively. To test the performance overhead of this implementation we used the same machine and configuration, with a different kernel version (2.6.27-9) and QEMU 0.9.1 (svn trunk). The performance of the same countermeasure implemented into hardware greatly increased, as expected (Table 2) We have performed a comparison with the original unprotected QEMU to have an idea of the overhead - up to 6% - introduced by our countermeasure with respect to the non instrumented version.

6 Future work

As previously reported our countermeasure does not detect if a buffer overflow has occurred since it overwrites the return address of the protected function, without checking if it has been altered.
Is a failed-with-segfault program better than one that repairs the fault and continues its execution as nothing happened? This is still an unanswered question. Our countermeasure allows the protected function to recover its caller's return address and continue its normal execution flow. We are aware that there are cases in which it is better to terminate the attacked program and log that a buffer overflow has occurred. A check of the saved return address to detect if the one currently on the stack has been tainted might be implemented with a little more overhead.
As explained this countermeasure protects only return addresses of functions. As discussed in [30], functions protected by such a countermeasure suffer from the attack to other components of the stack as local variables, function's arguments, saved frame pointers, etcetera. This countermeasure could be extended based on the observation that many exploits that result in code execution, rely on modifying the value of a pointer. In this extended countermeasure, all pointers are stored in read-only memory preventing them from being modified directly by a vulnerability. Whenever the program writes to a pointer, the compiler generates a special instruction that can be used to write to this pointer. If a vulnerability exists in a program that allows an attacker to overwrite arbitrary memory locations, this is prevented because only the special instruction can write to pointers. Since attackers are not be able to generate this special instruction until they are able to achieve code execution, this countermeasure would provide a significant improvement in protection against exploitation of these vulnerabilities.

7 Conclusion

We described a countermeasure against the return address attack and implemented it by emulating hardware instructions for a virtualizing environment. Our main goal was to implement such a countermeasure for the widely used virtualization technology of Xen. Macro and micro benchmarks revealed a high overhead mainly caused by the context switch needed to emulate the necessary instructions to store and read the return addresses from a protected page. This means that it is not a countermeasure that can be used in a realistic deployment setting. However, results from the same implementation for an emulated processor showed that it may be more feasible in hardware.
We have considered virtualization as a platform to develop countermeasures for. With a different approach, the idea of implementing countermeasures against buffer overflows in hardware and virtualization is a promising solution in terms of performances and security.

8 References

[1] *QEMU open source processor emulator*, `http://bellard.org/qemu`.

[2] *The Xen hypervisor, the powerful open source industry standard for virtualization*, `http://www.xen.org`.

[3] *Windows Server 2008*, `http://www.microsoft.com/windowsserver2008`.

[4] National institute of standards and technology, *National vulnerability database statistics*, `http://nvd.nist.gov/statistics.cfm`.

[5] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353, Alexandria, Virginia, U.S.A., November 2005. ACM.

[6] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, Oakland, California, U.S.A., May 2008. IEEE.

[7] S. Alexander. Defeating compiler-level buffer overflow protection. *;login: The USENIX Magazine*, 30(3), June 2005.

[8] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanović, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS2003)*, pages 281–289, Washington, D.C., U.S.A., October 2003. ACM.

[9] S. Bhatkar, D. C. Duvarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, Washington, D.C., U.S.A., August 2003. USENIX Association.

[10] S. Bhatkar and R. Sekar. Data space randomization. In *Proceedings of the 5th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, volume 5137 of *Lecture Notes in Computer Science*, Paris, France, July 2008. Springer.

[11] T. Chiueh and F. H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 409–420, Phoenix, Arizona, USA, April 2001. IEEE Computer Society, IEEE Press.

[12] T. cker Chiueh and F. hau Hsu. Rad: A compile-time solution to buffer overflow attacks.

[13] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, Washington, D.C., U.S.A., August 2003. USENIX Association.

[14] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, San Antonio, Texas, U.S.A., January 1998. USENIX Association.

[15] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceeding of the 28th international conference on Software engineering*, pages 162–171, Shanghai, China, 2006. ACM Press.

[16] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*, pages 69–80, San Diego, California, U.S.A., June 2003. ACM.

[17] U. Erlingsson. Low-level software security: Attacks and defenses. Technical Report MSR-TR-2007-153, Microsoft Research, November 2007.

[18] V. Glaume and P. A. Fayolle. A buffer overflow study: Attacks & defenses. Technical report, ENSEIRB, 2002.

[19] R. Grimes. Preventing buffer overflows in C++. *Dr Dobb's Journal: Software Tools for the Professional Programmer*, 29(1):49–52, January 2004.

[20] M. Howard and D. LeBlanc. Writing secure code, 2002.

[21] Intel. Intel virtualization technology. `http://www.intel.com/technology/computing/vptech`.

[22] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2003.

[23] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, California, U.S.A., June 2002. USENIX Association.

[24] L. John. Spec cpu2000: Measuring cpu performance in the new millennium.

[25] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging*, number 009-02 in Linköping Electronic Articles in Computer and Information Science, pages 13–26, Linköping, Sweden, 1997. Linköping University Electronic Press.

[26] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS2003)*, pages 272–280, Washington, D.C., U.S.A., October 2003. ACM.

[27] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, California, U.S.A., August 2002. USENIX Association.

[28] S. Kowshik, D. Dhurjati, and V. Adve. Ensuring code safety without runtime checks for real-time control systems. In *Proceedings of the International Conference on Compilers Architecture and Synthesis for Embedded Systems*, pages 288–297, Grenoble, France, October 2002.

[29] G. Necula, S. Mcpeak, and W. Weimer. CCured: Type-

safe retrofitting of legacy code. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, Portland, Oregon, U.S.A., January 2002. ACM.

[30] G. Richarte. Four different tricks to bypass stackshield and stackguard protection. *World Wide Web*, (1), 2002.

[31] J. S. Robin. Abstract analysis of the intel pentiums ability to support a secure virtual machine monitor.

[32] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, San Diego, California, U.S.A., February 2004. Internet Society.

[33] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 144–155, Oakland, California, U.S.A., May 2001. IEEE Computer Society, IEEE Press.

[34] Vendicator. Stackshield, *A stack smashing technique protection tool*, `http://www.angelfire.com/sk/stackshield`.

[35] J. Xu, Z. Kalbarczyk, S. Patel, and R. K. Iyer. Architecture support for defending against buffer overflow attacks. 2002.

[36] W. Xu, D. C. Duvarney, and R. Sekar. An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 117–126, Newport Beach, California, U.S.A., October-November 2004. ACM, ACM Press.

[37] Y. Younan. *Efficient Countermeasures for Software Vulnerabilities due to Memory Management Errors*. PhD thesis, Katholieke Universiteit Leuven, May 2008.

[38] Y. Younan, W. Joosen, and F. Piessens. Code injection in c and c++ : A survey of vulnerabilities and countermeasures. Technical report, Departement Computerwetenschappen, Katholieke Universiteit Leuven, 2004.

[39] Y. Younan, W. Joosen, and F. Piessens. Efficient protection against heap-based buffer overflows without resorting to magic. In *Proceedings of the International Conference on Information and Communication Security (ICICS 2006)*, volume 4307. Springer-Verlag, December 2006.

[40] Y. Younan, D. Pozza, F. Piessens, and W. Joosen. Extended protection against stack smashing attacks without performance loss. In *Proceedings of the Twenty-Second Annual Computer Security Applications Conference (ACSAC '06)*, pages 429–438. IEEE Press, December 2006.