Proc. of The Fourth Intl. Conf. On Advances In Computing, Control And Networking - ACCN 2016

ISBN: 978-1-63248-095-8 doi: 10.15224/ 978-1-63248-095-8-05

Hardware-Based Stack Smashing Attack Detection & Preliminaries on Recovery Procedure

Raphael Segabinazzi Ferreira, Fabian Vargas, Letícia Bolzani Poehls

Abstract - **In recent years, computer systems belonging to large companies, governments as well as personal computers have been experiencing an increasing wave of attacks that disrupt their normal operation or leak sensitive data. In this context, this paper presents a hardware-based approach (here namely a watchdog connected to the processor bus) which aims at detecting and recovering the system from stack smashing buffer overflow attack. Compared to existing approaches, the proposed technique does not need application code recompilation or use of any kind of supervisor software (e.g., an Operating System - OS) to manage memory usage. To validate the approach, a case-study based on the LEON3 softcore processor and benchmark test codes have been implemented. Experimental results indicate that this approach is able to detect and recover from an intrusion tentative for 100% of the test cases, while yielding low area overhead, negligible attack detection latency and processor performance degradation.**

Keywords - **Stack Smashing Buffer Overflow Attack, Malicious Code Injection, Attack Detection and Recovery, Dynamic Integrity Checking, Secure Embedded System.**

I. Introduction

The need to include security mechanisms in electronic devices has dramatically grown with the widespread use of such devices in our daily life. In this scenario this work presents a hardware-based approach (here defined as a watchdog connected to the processor bus) which aims at detecting and recovering the system from stack smashing buffer overflow attack. The detection is done when the execution program returns from a function call. In the sequence, this work proposes a way to securely recover the system from the intrusion detection event. The preliminaries of this work were first published in [1], where the detection mechanism was first introduced. The present paper represents a continuation of this work, by introducing the recovery process, after the intrusion detection event by the watchdog.

II. Preliminaries

a) Stack Smashing Buffer Overflow Attack:

Buffer overflow attacks exploit a lack of bounds checking on the size of input being stored in a buffer array in memory. By writing data *past* the end of an allocated array, the attacker can make arbitrary changes to program state stored adjacent to the array. By far, the most common data structure to corrupt in this fashion is the stack, called a “stack smashing” or “buffer overflow” attack.

Many C programs have buffer overflow vulnerabilities, both because the C language lacks array bounds checking, and because the culture of C programmers encourages a performance-oriented style that avoids error checking where possible [2].

The common form of buffer overflow exploitation is to attack buffers allocated on the stack. Stack smashing attacks strive to achieve two mutually dependent goals:

i) *Inject Attack Code*: The attacker provides an input string that is actually an executable binary code native to the machine being attacked. Typically, this code is simple and does something similar to exec(“sh”) to produce a root shell.

ii) *Change the Return Address:* There is a stack frame for a currently active function above the buffer being attacked on the stack. The buffer overflow changes the return address to point to the attack code. When the function returns, instead of jumping back to where it was called from, it jumps into the attack code.

b) Related Works:

Several efficient software-based as well as hardware-based dynamic integrity checking techniques [3,4] have been proposed in the literature. However, software-based techniques suffer from performance overheads as high as 60%, while hardware-based approaches result in average overheads of about 18% [5]. These are daunting numbers. Additionally, some of these approaches [5,6] need application code recompilation to compute specific information (hashes of application program’s instruction addresses and opcodes) that is later used at runtime to detect attacks.

III. The Proposed approach

This work proposes two assumptions (see Fig. 1): (1) a watchdog to detect malicious data overwritten in the return addresses saved in the program stack, and (2) a method to recover the system from the return address overwritten detection. The overwritten activity occurs in buffer overflow conditions and could be the tentative of a stack smashing attack. So, this approach is able to detect a tentative of attack and recover the system from this condition into a safe point (by means of a checkpoint previously saved during normal execution).

Raphael Segabinazzi Ferreira, Fabian Vargas and Letícia Bolzani Poehls are with the Catholic University – PUCRS. Electrical Engineering Dept.
Av. Ipiranga 6681.
90619-900, Porto Alegre, Brazil.

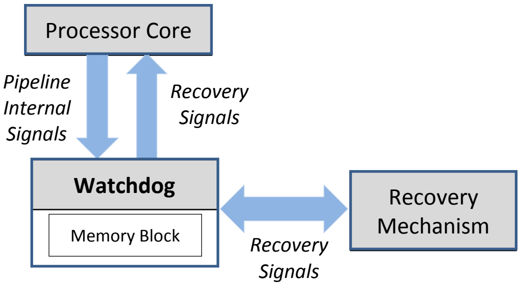

Fig. 1. General architecture of the proposed approach.

The proposed watchdog is based on two specific structures: (a) the logic implemented in hardware to detect the return address overwritten and (b) the memory block added to the Watchdog architecture to store return addresses of system calls. In more detail, this approach works as follows:

- Every time a call ("CALL") instruction is executed by the processor, the return address is stored in the original stack (typically a memory address or a dedicated register inside the processor) and in the Watchdog Memory Block;
- Every time a return instruction is executed, the Watchdog performs a comparison between the return address stored in the original stack and the return address stored in the Watchdog Memory Block. In this case, one of these two situations may occur:
  - In case of a *positive* comparison, the fault-free execution of the code continues and no action is taken by the Watchdog;
  - In case of a *negative* comparison, the Watchdog raises a signal to the Recovery Mechanism.

Therefore, in case of occurring an overflow on the original stack that corrupts the current return address, such address remains unchanged in the Watchdog Memory Block. This condition guarantees the detection of tentative of intrusion by comparing the return address overwritten in the original stack against the (fault-free) return address stored in the Watchdog Memory Block. In the sequence, the Watchdog does not allow the system to branch to any possible malicious code pointed by the corrupted return address and then, raises a signal to trigger the recovery process.

Starting from the signal generated by the Watchdog, the recovery algorithm rollbacks the processor to the last safe-checkpoint generated previously in run-time. This safe point is the correct return address of the function CALL whose "*RET*" address was object of attack. When the rollback process is complete, the Watchdog releases the processor to run again normally.

Given the above exposed, the proposed approach presents the following features and advantages compared to the existing techniques:

- It does not need application code recompilation to compute specific information (hashes of application program's instruction addresses and opcodes) that is later used at runtime to detect attacks.
- It is not based on any software component and considering that the Watchdog works in parallel with processor execution, the proposed approach does not incur in performance overhead.
- The watchdog detection mechanism requires a low area overhead.
- Extremely low attack detection latency, since the Watchdog works in parallel with the processor execution.

a) Attack Detection

As observed in Fig. 1, the Watchdog monitors some internal signals from the *Execution Stage* of the processor pipeline. Such signals are:

- The "*OpCode*" of the instruction that is leaving the Execution Stage of the pipeline;
- The bit "*annul*", whose function is to indicate if the instruction that is leaving the Execution Stage of the pipeline will be actually executed by the processor or it will be discarded due to speculative execution.
- The "*Program Counter*" (PC), which is saved into the *ShadowStack* in case a function "CALL" is performed. After the function execution, the PC is defined as the return address that will be used to return processor control to the point where the application was interrupted.
- The "*jmp_addr*" signal, which points to the function return address that will be executed.

Fig. 2 shows the internal blocks of the Watchdog. As detailed above, it grabs a set of 4 specific pipeline internal signals. The *Instruction Decoder* Block uses the instruction "*OpCode*" and the "*annul*" signal to decode and check if the current instruction will be executed. If the *Instruction Decoder* Block decodes a function "CALL", it will send the "*icall*" signal to the *ShadowMem Control* Block. In this case, the *ShadowMem Control* Block will save the current PC retrieved from the pipeline ("*Curr_PC*" signal) into the *ShadowMem* Block. Instead, if the *Instruction Decoder* Block decodes a function "RET", it will send the "*ijmp*" signal to the *ShadowMem Control* Block that will recover from the *ShadowMem* Block the last PC saved therein and send it (in conjunction with a "*compare*" signal) to the *Decision* Block.

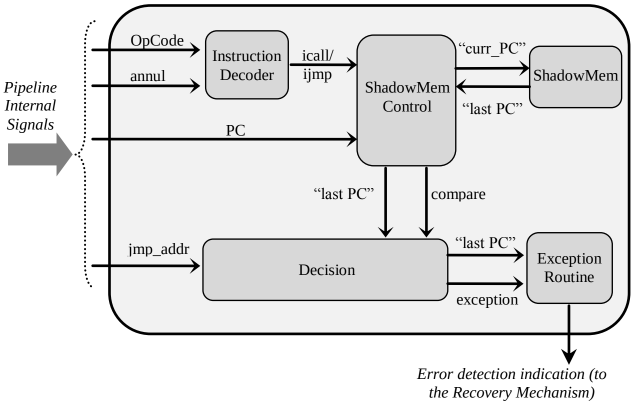

Fig. 2. Internal blocks of the Watchdog.

When the *Decision* Block receives the "*compare*" signal and the "*last PC*", it performs a comparison between this value (the "*last PC*") and the "*jmp_addr*" retrieved from the pipeline. If this comparison returns *true*, no action is required. Nevertheless, if the comparison returns *false*, the "last PC"

Proc. of The Fourth Intl. Conf. On Advances In Computing, Control And Networking - ACCN 2016
Copyright © Institute of Research Engineers and Doctors, USA .All rights reserved.
ISBN: 978-1-63248-095-8 doi: 10.15224/ 978-1-63248-095-8-05

and an "*exception*" signal are sent to the *Exception Routine* Block.

When the *Exception Routine* Block receives the "exception" signal, it raises a signal to the Recovery Mechanism Block to trigger the recovery process. At this point, the recovery process is based on a dedicated algorithm that rolls back the processor to the last safe-checkpoint labeled by the Recovery Mechanism Block, concurrently with processor execution. This checkpoint is the correct return address of the function CALL whose "*RET*" address was object of attack). When the rollback process is complete, the Watchdog releases the processor to run again normally.

b) Attack Recovery

The recovery process can be divided into two steps: the *Regular Execution* and the *Recovery Process* itself.

b.1) Regular Execution (before attack detection):

When the program is running on the *main* function, the execution data is saved in the original memory (as in any conventional processor) and in the Secondary Memory settled in the Recovery Mechanism Block. When processor execution leaves the *main* function by means of a function CALL:

- The processor continues saving the program data into the original memory.
- The Watchdog stops saving data in the Secondary Memory and starts saving program data on the Recovery Memory which is also sitting in the Recovery Mechanism Block. The Watchdog also saves in this memory the address where this (original) data is saved in the original memory. These data are saved in a queue order.
- The Watchdog also saves the states of all processor general purpose and control registers in the Recovery Memory in order to build-up a safe check-point for eventual recovery process.

When the execution returns to *main* function the Watchdog performs the following actions:

- The program data temporarily stored in the Recovery Memory are committed to the Secondary Memory in a FIFO (first in – first out) order.
- The stack register states saved in the Recovery Memory are discarded. The data saved in this step are just the data owned by the piece of stack that was not released yet by the program execution and the data in other memory spaces with write permission (*e.g.*, heap and data).
- And the processor continues normal execution.

b.2) Recovery Process (upon attack detection):

When the Watchdog detects a return address overwritten it raises the *Error detection* signal (Fig. 2) to the Recovery Mechanism Block, which in turn performs the following steps:

- The program execution is stopped.
- The system is rolled back to the most recent safe-checkpoint labeled by the Recovery Mechanism Block in run-time. This safe point is the correct return address of the function CALL whose "*RET*" address was object of attack. So, the registers states saved in the Recovery Memory will be attributed by the Recovery Mechanism Block to the processor (general purpose and control) registers.
- All data saved in the Recovery Memory (including the stack registers data) will be discarded.
- The Recovery Mechanism Block overwrites the original memory with data stored in the Secondary Memory, and
- Finally, the program restarts running.

If a second function CALL is executed before returning from the current function under execution (nested function CALL) then the approach stops saving program data in the current Recovery Memory (for instance, assume this as the Recovery Memory - Level 1) and switches to save data (including all processor general purpose and control registers) into another memory area (let us say, Recovery Memory - Level 2) and so successively. Similarly, when the processor returns from the second function CALL to the function that called it (first function CALL) the program data temporarily stored in the Recovery Memory – Level 2 are committed to the Recovery Memory – Level 1 and the stack register states saved in the Recovery Memory – Level 2 are discarded.

In the event of attack detection, the processor is rolled back to the immediate lower Recovery Memory Level. For instance, assume that the processor is returning from a function CALL where all program data and processor general purpose and control registers are being saved in the Recovery Memory – Level 2, then the registers states saved in the Recovery Memory – Level 2 will be attributed by the Recovery Mechanism Block to the processor (general purpose and control) registers. Next, all data saved in the Recovery Memory – Level 2 (including the stack registers data) are discarded, the Recovery Mechanism Block overwrites the original memory with data stored in the Recovery Memory – Level 1, and the program restarts running from the first function CALL.

IV. Experimental Results

This approach was implemented on the LEON3 softcore processor [7]. The LEON3 is a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture. The model is highly configurable, and particularly suitable for system-on-a-chip (SoC) designs. Fig. 3 depicts the general block diagram of the processor core. Blocks indicated by (*) are optional and are included in the processor main architecture if selected by the designer. Therefore, the basic processor configuration is the LEON3 CPU Integer Unit, the AMBA AHB Master Interface and the AMBA Bus, which connects the CPU to the system memory.

Table 1 shows the area overhead added by the watchdog implementation. This table depicts results for two different implementations of the watchdog according to its ability to monitor and capacity to store nested function calls: in the first implementation, the watchdog is able to handle 256 function calls and return addresses, while in the second implementation it supports the monitoring and storage of 64 return addresses.

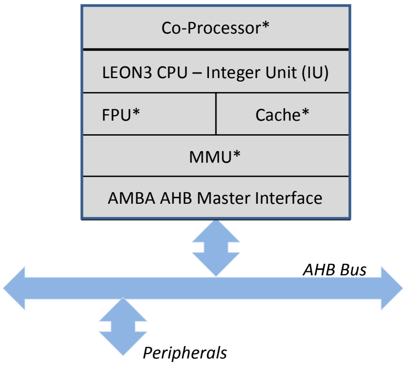

Fig. 3. General block diagram of the LEON3 softcore processor.

Table 1. Area overhead yielded by the Watchdog implementation.

Return Address Capacity	Leon3 + Watchdog [nº primitives]	Watchdog entity [nº primitives]	Area Overhead [%]
256	13402	1016	7.58
64	12997	611	4.70

To validate this approach a simple C program was implemented. This program performs a buffer overflow that overwrites a return address located in the stack. As expected, when the processor tries to execute the return address, the watchdog immediately generates the exception signal due to return address overwritten. In the sequence, the recovery mechanism successfully rollbacks the processor to the most recent saved checkpoint.

To make a more severe analysis of the approach, benchmark test programs were implemented with pieces of known vulnerable C codes. These vulnerable pieces of C code were obtained from vulnerable test benchmarks published in the CVE (Common Vulnerabilities and Exposures) [8]. These code snippets were adapted and included into the test program source codes. Then, while running these programs the Watchdog and the Recovery Mechanism successfully generated the exception signal and recovered system from the unsafe state.

Finally, we have also checked the attack detection latency of the Watchdog. More precisely, we measured the time between the instant at which the Watchdog detects that an incorrect return ("*RET*") instruction will be executed in the processor pipeline and the instant at which the correct return address of the function CALL whose "*RET*" address was object of attack is copied into the Program Counter: $PC = CorrectRetAddress$. In this case, the measured attack detection latency was 4 clock cycles.

V. Final Considerations

This paper presented a hardware-based approach to protect systems from stack smashing attacks. Additionally, we have presented a preliminary discussion on the recovery process for the system from the unsafe state by rolling-back the processor to the last saved checkpoint.

Experimental results show that this approach successfully detected and recovered 100% of the injected attacks under the analyzed situations. The approach yields a low area overhead associated with extremely low attack detection latency.

Currently, we are implementing a complete case study in order to estimate the area overhead induced by the Recovery Block, for different levels of nested functions CALLs.

Acknowledgment

This work has been supported in part by CNPq (National Science Foundation, Brazil) under contract n. 303701/2011-0 (PQ) and Hewlett-Packard Brazil Ltd. using incentives of Brazilian Informatics Law (8.2.48 from 1991).

References

[1] R. Segabinazzi Ferreira, F. Vargas. "ShadowStack: A new approach for secure program execution", Microelectronics and Reliability Journal, 55(9) August 2015, pp. 2077-2081.

[2] B. P. Miller, D. Koski, C. Pheow Lee, V. Maganty, R. Murthy, A. Natarajan, J. Steidl. "Fuzz Revisited: A Reexamination of the Reliability of UNIX Utilities and Services", Report: University of Wisconsin, 1995.

[3] M. L. Corliss, E. C. Lewis, A. Roth, "Using DISE to Protect Return Addresses from Attack", Workshop on Architectural Support for Security and Anti-Virus (WASSA), Oct. 2004.

[4] Y. Park., Z. Zhang, G. Lee, "Microarchitectural Protection Against Stack-Based Buffer Overflow Attacks", IEEE Micro, vol. 26 , issue 4, July-Aug. 2006.

[5] A. K. Kanuparthi, R. Karri, G. Ormazabal, S. Addepalli, "A High-Performance, Low-Overhead Microarchitecture for Secure Program Execution", IEEE International Conference on Computer Design (ICCD), Oct 2012, Montreal, Canada.

[6] M. A. Schuette, J. P. Shen, "Processor Control Flow Monitoring Using Signatured Instruction Streams", IEEE Transactions on Computers, vol. 36, no. 3, March 1987, pp. 264-276.

[7] URL: http://gaisler.com/index.php/products/processors/leon3. Last visit: April 2016.

[8] Common Vulnerabilities and Exposures - The Standard for Information Security Vulnerability Names. URL: https://cve.mitre.org/. Last access: April 2016.