ShadowStack: A new approach for secure program execution

Raphael Segabinazzi Ferreira, Fabian Vargas *

Electrical Engineering Faculty, Catholic University — PUCRS, Porto Alegre, Brazil

1. Introduction

The need to include security mechanisms in electronic devices has dramatically grown with the widespread use of such devices in our daily life. With the increasing interconnectivity among devices, attackers can now launch attacks remotely. Such attacks arrive as data over a regular communication channel and, once resident in the program memory, they trigger a pre-existing software flaw and transfer control to the attacker’s malicious code. Software vulnerabilities have been the main cause of computer security incidents. Among these, buffer overflows are perhaps the most widely exploited type of vulnerability, accounting for approximately half the CERT advisories in recent years [1].

In this scenario, this paper presents a new hardware-based approach to detect stack smashing buffer overflow attack. This approach does not need application code recompilation or use of any kind of software (e.g., an OS) to manage memory usage. According to preliminary implementations, this approach guarantees 100% attack detection, while resulting in negligible area overhead and zero performance degradation (since the watchdog is fully independent from the processor and performs in parallel to the code execution).

2. Preliminaries

2.1. 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 [7,8].

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:

a) Inject attack code: The attacker provides an input string that is actually 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.

b) 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 to the attack code.

2.2. Related works

Several efficient software-based as well as hardware-based dynamic integrity checking techniques [14–16] have been proposed in the literature. However, software-based techniques suffer from performance overheads as high as 60%, while hardware-based approaches result

* Corresponding author.
E-mail address: vargas@computer.org (F. Vargas).

Please cite this article as: R.S. Ferreira, F. Vargas, ShadowStack: A new approach for secure program execution, Microelectronics Reliability (2015), http://dx.doi.org/10.1016/j.microrel.2015.07.021
in average overheads of about 18% [3]. Additionally, some of these approaches [3,9] need application code recompilation to compute specific information (hashes of application program's instruction addresses and opcodes) that are later used at runtime to detect attacks (Table 1).

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

- does not need application code recompilation;
- it is not based on any software component;
- low area overhead and negligible performance degradation; and
- extremely low attack detection latency.

3. The proposed approach: ShadowStack

The proposed approach is based on two specific structures: (a) the implementation of a watchdog in hardware and (b) on the reservation of a dedicated memory space that is used to store the return addresses of functions. In more detail, the ShadowStack approach works as follows:

- every time a call (CALL) function 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 ShadowStack; and
- every time a return instruction from a function is executed, the watchdog performs a comparison between the return address stored in the original stack and the ShadowStack. In this case, one of the two situations may occur:
  - in case of a positive comparison, the regular execution of the code takes place; and
  - in case of a negative comparison, a recovery routine is executed by the processor as response to a possible attack to the system. This recovery routine copies the return address stored in the ShadowStack into the original stack to be read by the processor. At the end of this process, the recovery routine returns processor execution to the application code.

Therefore, in case of occurring an overflow of the original stack that corrupts the return address, such address remains unchanged in the ShadowStack register. This condition guarantees detection of the execution of an invalid instruction by means of comparing the copies of the return address stored in both stacks.

Fig. 1 depicts the general block diagram of the proposed ShadowStack Approach, where the watchdog is instantiated besides the processor core. As observed in Fig. 1, the watchdog monitors some internal signals from the execution stage of the processor pipeline. Such signals are as follows:

- 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.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Advantages</th>
<th>Drawbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>StackGuard [5,6,11]</td>
<td>- Independent of the HW (processor architecture) or SW (application code description language and OS) platforms.</td>
<td>- Need application code recompilation.</td>
</tr>
<tr>
<td>Dynamic integrity checking [3,4]</td>
<td>- High detection coverage</td>
<td>- Based on SW, thus implying considerably performance degradation.</td>
</tr>
<tr>
<td>Light-weight hardware return address and stack Frame Tracking [10]</td>
<td>- Added stack in hardware</td>
<td>- Need an operating system (OS) to manage memory.</td>
</tr>
<tr>
<td>Separates the stack to two parts [17]</td>
<td>- Independent of the HW/SW platform</td>
<td>- Need application code recompilation.</td>
</tr>
</tbody>
</table>

Fig. 1. Watchdog instantiated besides the processor core.

- 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 proposed 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.

When the Decision Block receives the “compare” signal and the “last PC”, it performs a comparison between this value 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” and an “exception” signal are sent to the Exception Routine Block.

When the Exception Routine Block receives the “exception” signal, it generates an interruption to the processor core. This interruption forces the processor to jump to a specific memory area to execute a recovery routine or in the worst case, to reset the processor. Note that the uncorrupted function return address is stored in the Exception Routine Block and can eventually be retrieved by the recovery routine if roll-forward strategies are considered.

The flow chart shown in Fig. 3 describes the basic actions performed by the watchdog.

By monitoring processor pipeline internal signals, it is possible to detect the moment when a “CALL” instruction is executed. At this moment, the current PC is simultaneously saved in the regular stack (stack pointer: SP) and in the ShadowStack (ShadowStack pointer: SSP). In more detail, the following operations are performed:

\[
\text{Stack}[: \text{SP} + 4] = \text{PC} \\
\text{ShadowMem}[: \text{SSP} + 4] = \text{PC}.
\]
After saving PC context in SP and SSP, both stacks are incremented as follows:

\[
SP = SP + 4 \\
SSP = SSP + 4.
\]

When a “RET” instruction is executed by the processor, both stacks are read out by the watchdog and their contents compared:

\[\text{Stack}(SP-4) = \text{ShadowMem}(SSP-4).\]

If the comparison is true, the PC register receives the PC stored in the original stack and the stack pointer (SP) and the ShadowStack pointer (SSP) are updated with the next return address, as follows:

\[
PC = \text{MEM}[SP-4] \\
SP = SP - 4 \\
SSP = SSP - 4.
\]

4. Experimental results

The watchdog was implemented on a LEON3 softcore processor [12]. LEON3 is a synthesizable VHDL model of a 32-bit 7-stage pipeline processor compliant with the SPARC V8 architecture. The model is highly configurable, and particularly suitable for system-on-a-chip (SOC) designs. The full source code is available under the GNU GPL license, allowing free and unlimited use for research and education. LEON3 is also available under a low-cost commercial license, allowing it to be used in any commercial application to a fraction of the cost of comparable IP cores. The ISE-Xilinx design framework was used to simulate, configure and evaluate the proposed approach (LEON3 + watchdog) on a Virtex-4 (XC4VFX12-10SF363) FPGA. Table 2 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.

To validate the watchdog a simple C program was implemented. This program performs a buffer overflow that overwrites a return address located in the stack. The C source code of this program is shown below:

```c
void sploit(void)
{
    char buff[8];
    memset(buff, 0xff, 1024);
    return;
}
void sploit1(void)
{
    sploit();
    return;
}
int main(void){
    sploit1();
    return 0;
}
```

This program uses the memset function to set a value in the char array called “buff”. The memset function fills the first 1024 bytes of the memory area pointed to by buff with the constant value “0xff”. However the size passed as parameter (1024 bytes) is much larger than the char array buff size (only 8 bytes). So, as explained in Section 2, we have a buffer overflow situation. In this case, the sploit1 function return address (which was saved in the stack) is overwritten by the value passed by the memset function. Finally, as expected, when the processor tries to execute the return address, the watchdog immediately generates the exception signal due to return address overwritten.

To make a more severe analysis of the watchdog functionality, 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) [13,18]. These code snippets were adapted and included into the test program source codes. Then, while running these programs the watchdog was evaluated and the obtained results are depicted in Table 3.

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 a “RET” instruction will be executed...
the four signals described in Section 3 can be retrieved (“Opcode”, “null”, “PC” and “jmpl_addr”). Concerning COTS processors (for instance Intel, PowerPC and ARM), it is probably more difficult to monitor pipeline internal signals, but this can be done by using the JTAG interface.

(b) Considering the detection coverage of the watchdog, our solution can be extended to processors that use any instruction to return from functions (LEON3 does not use the RET instruction too, but a jmpl instruction in especial conditions). So, we just need a way to differ these instructions from any other in the processor instruction set.

Moreover, assume for instance that a link register is used, then the PC is not stacked but the linker register is. In this case, we obtain the return address when a function is called (i.e., the exact moment when the instruction that calls the function is being executed), then we grab the return address from the current PC. From the watchdog point of view, it does not matter if this data will be saved first in the link register and then moved to the stack or saved directly from the PC to the stack.

Finally, the watchdog is not capable to detect pieces of code containing function CALLs that do not have a RET instruction or indirect function CALLs (for instance, pointers used to call functions). In this case, a tentative of intrusion would not be detected by the watchdog. Having this issue in mind, we are currently working in a set of improvements to the proposed technique that will allow the watchdog to detect not only this type of intrusion, by a few others. Nevertheless, note that the proposed watchdog is assumed to be used in critical applications, which by nature do not use function pointers in order to satisfy safe software design practices [19].

6. Final considerations

This paper presented ShadowStack, a new dynamic integrity checking technique based on a watchdog implemented in hardware. The watchdog observes specific instructions in the code being executed through the processor pipeline, compares them against reference values generated at runtime and in the event of detecting a tentative of intrusion, the pipeline is stalled and the instructions are not allowed to commit by flushing them from the pipe. The attack type treated in this work is stack smashing buffer overflow. Compared to the existing approaches found in the literature, the ShadowStack advantages are listed as follows: (a) does not need application code recompilation; (b) it is not based on any software component; (c) low area overhead and negligible performance degradation; and (d) extremely low attack detection latency.

Experimental results obtained throughout simulations of test programs that were implemented with pieces of known vulnerable C codes obtained from vulnerable test benchmarks published in the CVE (Common Vulnerabilities and Exposures) demonstrate that the technique is very efficient: so far, the totality of the simulated intrusions were detected by the watchdog. Furthermore, area overhead was measured against the implementation of a system based on the LEON3 softcore processor plus the watchdog, both described in VHDL and mapped into a Xilinx Virtex-4 FPGA. In this scenario, the observed area overhead was less than 1% of the LEON3 processor. Finally, the attack detection latency of the watchdog was measured to be very low: 6 clock cycles.
Table 3
Watchdog detection using pieces of vulnerable codes obtained from vulnerable test benchmarks published in CVE.

<table>
<thead>
<tr>
<th>Vulnerable programs</th>
<th>CVE number</th>
<th>Severity [18]</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edbrowse</td>
<td>CVE-2006-6909</td>
<td>10.0 high</td>
<td>Watchdog generates the exception signal</td>
</tr>
<tr>
<td>MADWiFi</td>
<td>CVE-2006-6332</td>
<td>7.5 high</td>
<td>Watchdog generates the exception signal</td>
</tr>
<tr>
<td>OpenSER</td>
<td>CVE-2006-6749</td>
<td>9.3 high</td>
<td>Watchdog generates the exception signal</td>
</tr>
<tr>
<td>Samba</td>
<td>CVE-2007-0453</td>
<td>4.6 medium</td>
<td>Watchdog generates the exception signal</td>
</tr>
<tr>
<td>Sendmail</td>
<td>CVE-2003-0681</td>
<td>7.5 high</td>
<td>Watchdog generates the exception signal</td>
</tr>
<tr>
<td>Wu-ftpd</td>
<td>CVE-1999-0368</td>
<td>10.0 high</td>
<td>Watchdog generates the exception signal</td>
</tr>
<tr>
<td>Wu-ftpd</td>
<td>CVE-2003-0466</td>
<td>10.0 high</td>
<td>Watchdog generates the exception signal</td>
</tr>
</tbody>
</table>

References


Please cite this article as: R.S. Ferreira, F. Vargas, ShadowStack: A new approach for secure program execution, Microelectronics Reliability (2015), http://dx.doi.org/10.1016/j.microrel.2015.07.021