Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 21 (2013) 250 – 256

Procedia Computer Science

The 4th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2013)

Stack Memory Buffer Overflow Protection Based on Duplication and Randomization

Sahel Alouneh[1], Mazen Kharbutli[2], Rana AlQurem[2]

[1]German Jordanian University, [2]Jordan University of Science and Technology,

sahel.alouneh@gju.edu.jo, kharbutli@just.edu.jo, rana_alqurem@yahoo.com

Abstract

With software systems continuously growing in size and complexity, the number and variety of security vulnerabilities in those systems is increasing in an alarming rate. Vulnerabilities in the program's stack are commonly exploited by attackers in the form of stack-based attacks. In this paper, a software based solution for stack-based vulnerabilities and attacks is proposed and implemented. The proposed solution involves creating a new patch tool that fixes a wide-range of stack related vulnerabilities in the existing applications. The basic idea of our approach is to implement a patch tool that makes multiple copies of the return addresses in the stack, and then randomizes the location of all copies in addition to their number. All duplicate copies are updated and checked in parallel such that any mismatch between any of these copies would indicate a possible attack attempt and would trigger an exception. The results of our implementation show high protection against integer overflow and buffer overflow attacks.

Selection and peer-review under responsibility of Elhadi M. Shakshuki

Keywords: Stack-based protection; Buffer overflow; Security

1. Introduction

Nowadays, security is one of the major concerns for computer users. Computer attacks usually exploit vulnerabilities in the software applications that take focus on space and performance in their design consideration over safety.

Buffer overflow in the program's stack is a very common vulnerable place for those attacks to be applicable [1]. The main objective of the attacker is usually to change the control flow of the program, allowing the attacker to execute arbitrary code such as opening a new shell with the same access rights to

* Corresponding author. Tel.: +96264294122; fax: +96264294122.
E-mail address: sahel.alouneh@gju.edu.jo

Selection and peer-review under responsibility of Elhadi M. Shakshuki
doi:10.1016/j.procs.2013.09.033

the system as of that process attacked. If the process has a root access, so will the attacker in the new shell, leaving the whole system open for any kind of manipulation [2].

Buffer overflow attacks make up approximately half of all software security vulnerabilities [3, 4]. According to statistics from the Computer Emergency Response Team Coordination Center at Carnegie Mellon University (CERT) [5], the number of reported vulnerabilities in software has increased by a factor of about 10x in the period between year 2000 and 2012 as shown in Figure 1.

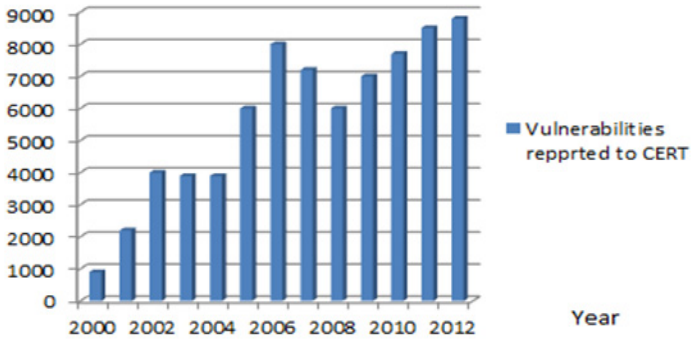

Fig.1. Software vulnerabilities reported to CERT

Such attacks are easily performed as evident by their widespread compared to other forms of attacks. Since most of stack based buffer overflow attacks lead to a compromised return address, our proposed solution focused on protecting the stack from attacks against the return addresses. Many approaches have been proposed to protect against stack based attacks, especially to prevent compromised return address. The diversity of all previous proposed mechanism can be categorized into two main different solutions: Software-based [7-16] and Hardware-based [17- 22]. Software solutions focus on existing source code while Hardware solutions need major changes in the architecture.

This paper is organized as follows: Section 2 presents a literature review for solutions that are related to our work and also compare them with our proposed solution. Our proposed solution and the implementation are discussed in Section 3. The simulation results and evaluation of our software solution is presented in Section 4. Finally, the discussion in Section 5 summarizes and concludes the paper.

2. Related Work

In this Section we present and discuss the main proposed approaches that protects against stack based attacks, and especially in preventing the compromise of the return address. The defense techniques are categorized into two main types: software based or hardware based solutions.

2.1. Software Solution

Software defense mechanisms are widely used and more attractive for computer users since it is mostly independent of operating system and it does not require adding new hardware or changing in the architecture. Most of available software defense mechanisms that protect the stack against buffer overflow attacks are implemented as compiler extensions. Therefore, they require the source code of the vulnerable programs so they can be re-compiled. This requirement is very difficult in some cases especially with legacy applications and commercial software. Examples of software based solutions are found in references [7, 8, 9, 10, 11, 12, 13, 14, 15, and 16].

2.2. Hardware Solution

Although hardware solutions may show more resistance against buffer overflow attacks in some cases, it is not a favorable by computer users. All current hardware solutions require a specific hardware or configuration or the both to be implemented. Examples of hardware based solutions are found in references [17, 18, 19, and 20].

3. Approach

In this paper, we propose a general software solution for stack-based vulnerabilities and attacks. Our goal is to provide a protection and at the same time maintaining high performance and lowest cost.

We create a new patch tool that fixes a wide-range of stack related vulnerabilities in existing applications. Mainly, our solution depends on duplication and randomization of the new stack locations where the return addresses will be stored and checked. Our solution is able to detect and prevent any type of attacks against return addresses on the stack. The idea behind our proposed solution is to create random number of copies of the return address on the stack, then allocating random stack location for each copy of the return address, and then start storing these copies from random offset within each allocated stack in each function prologue. Before the function return, all duplicate copies are checked in parallel such that any mismatch between any of these copies would indicate a possible attack attempt and would raise an exception. At run time, the secure program will create random number of instances of return addresses in the stack, called Mirror Return Address Stacks (MRAS), in the heap memory. The number of copies is (3 to 5) and locations of these MRAS structures is randomized based on the process ID, date and time, machine ID. Therefore, the number and locations of these MRAS structures will differ between different instances of the same program. When a return address is to be pushed onto the stack, copies are also pushed onto the MRAS structures. When the return address is to be used, all copies are checked for integrity.

3.1. Algorithm

Our proposed approach involves creating a new patch tool that fixes stack related vulnerabilities in existing binaries. The input is an assembly file which is extracted from the vulnerable executable binary file using an appropriate disassemble tool. The output is an enhanced assembly file with improved stack protection and security. This assembly file can then be changed back into an executable binary using the assembler. Figure 2 shows the main procedures to implement our proposed solution from vulnerable binary file to more secure binary file passing through three steps.

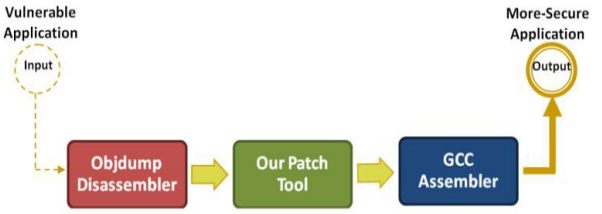

Fig. 2. Main steps to implement our proposed solution

3.2. Implementation of our Patch Tool

We create a new patch tool that fixes a wide-range of stack related vulnerabilities in existing applications. The input to our patch tool is an assembly file which is disassembled from the vulnerable binary file using *objdump* disassembler. The output is an optimized assembly file with improved stack protection and security. This assembly file can then be re-compiled into an executable binary using the assembler. Our new patch tool is implemented and written to work as summarized in Figure 3.

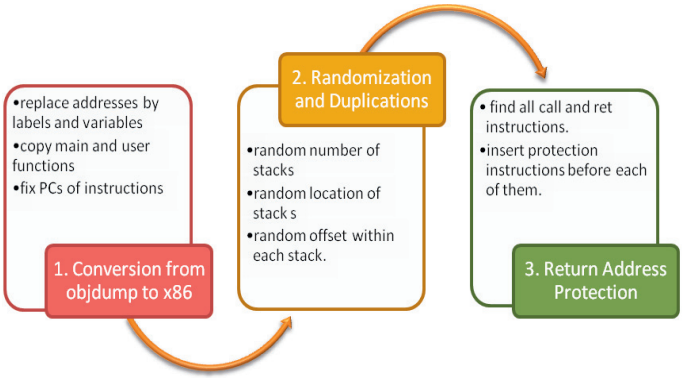

Fig .3. The implementation steps of our patch tool.

As we discussed above, the input to our patch tool is the output assembly file form Objdump disassembler. At this point, we faced a difficult problem since the output assembly file of objdump is not suitable for the purpose of compilation, so it cannot be used to re-create a binary file. Therefore, we convert the assembly file of objdump format to an x86 assembly file which is valid for compilation. The conversion has been done on several stages. Since objdump uses the address of variable instead of its name as operands in the assembly instructions, and x86 uses the name of variable, we should replace the address of variable by its name. The first stage of conversion involves storing the variables of BSS and TEXT segments that exist in dynamic table. The variables are saved in linked list called VARIABLE. Each address that used in assembly instruction during the code is compared with all addresses that stored in VARIABLE linked list and if it exists then the address is replaced by the name of variable of matched address. For the variables of BSS segment only, we insert a definition for each variable at the end of the output file that contains the .comm keyword along with the variable name, size, and alignment.

In the second stage, we scanned the input file to look for the section that includes all important labels that are used by the instructions during the code. This section in Objdump file is called .rodata section. The column on left-most contains the start addresses of .rodata section. In x86 assembly file, each label has a unique ID, therefore we used counter of 3 digits for labels IDs starting with LC000. To derive the needed labels for x86 assembly file from .rodata section in objdump file, we save the first start address as START_ADDRESS variable and the last address as END_ADDRESS which will be used later. In addition, the hexadecimal values are stored in HEXA_VALUE array and the whole characters in the right-most column are stored in the DATA_SECTION array. Figure 4 shows the labels in x86 assembly file that is derived from .rodata section.

```
LC000:
        .string  "the original matrix is :"
LC001:
        .string  "(%d,%d)=%f\t"
LC002:
        .string  "THE DETERMINANT OF THE ABOVE
MATRIX IS: %f \n"
LC003:
        .string  "the result matrix is:"
LC004:
        .string  "(%d,%d)=%f\t"
LC005:
        .long            0
        .long            10745291
LC006:
        .long            0
        .long            -6320914
```

Fig.4. The Labels in x86 assembly file.

As it has been discussed in the second stage, when the data section is converted to labels, four variables are saved:

- START_ADDRESS is the start address of characters in data section.
- DATA_SECTION array of type char contains all characters in data section
- HEXA_VALUE array of type char contains all Hexadecimal values in data section.
- LABEL_INDEX array of type *int* contains the index where the first character in each label is stored in DATA_SECTION array.

It is worth to note that our patch tool solves the problem of compilation by converting the input file from objdump format that cannot be compiled into an x86 compatible assemblers. Nevertheless, our generated assembly file is still vulnerable to stack-based attacks and no protection is provided yet to the input application. As we discussed earlier in this paper, stack-based attacks exploit vulnerabilities in software programs such as buffer overflow and especially programs written in C/C++ language since it does not perform boundary checking of arrays. Such attacks aim to overwrite the return address by an address that points to a malicious code which may cause serious problems. In this phase, our patch tool provides the generated x86 assembly file from previous phase with three levels of security to protect return addresses comprehensively on stack against all types of return address attacks. These security levels are based on randomizing and duplicating the locations where the return addresses will be stored. At run time, a random number between (1 and 5) is saved in nCopies variable. Then, nCopies of instances of return address stacks are created, called Mirror Return Address Stacks (MRAS), in the heap memory using sbrk( ) system call. Every MRAS has size of 10k bytes. Finally, the offset of each MRAS is also randomized such that the start location to store return addresses is different among all mirror stacks. The number nCopies, locations, and offsets of these MRAS structures are randomized based on the process ID, date and time, machine ID, etc. Therefore, the number, locations, and offsets of these MRAS structures will change between different instances of the same program.

4. Evaluation

As a proof of concept, our software tool is tested using several known programs to verify the performance overhead. In next section we used the most common stack based attacks to verify the correctness and the level of protection of our SOFTWARE TOOL. To test the performance overhead we ran several micro benchmarks. We collected the results of running programs instrumented with and without the code that implements the stack protection MRAS. All tests were run on a single machine (Intel(R) Core(TM)2 Duo CPU T5750@2.00GHz, 3064MB RAM, GNU/Linux fedora9). The GCC

compiler has been used to instrument assembler code with new instructions. The benchmarks show that this implementation experiences a significant slow-down factor of between 1% to 170% as shown in Table 1. The result of run time in Table 1 is the average of 50 runs for each program. There was no need to do more run because they were close in 1% to each other. The memory overhead of our tool is in range of 30 to 50 KB such we create 3 to 5 new stacks at size of 10 KB to protect return address.

Micro Benchmark	Runtime without MRAS(s)	Runtime without MRAS(s)	Overhead
Bubble Sort for 100000 numbers	72.063	73.462	1%
Sorting Algorithm for 20000 number by 6 algorithms	53.256	54.073	1%
Factorial for 10000000 numbers	109.851	109.874	16%
Matrix Inverse of 11x11	49.127	74.427	51%
Fibonacci for 100000 numbers	54.661	146.623	170%

Table 1. Micro Benchmarks results of our implementation

The MRAS implementation is affected by different increase of the execution time when compared to reference time in different programs as shown in Figure 5. That is mostly due to the number of function calls in the program such that the overhead is increasing as the number of function calls is increasing in program. In addition, the programs that contain recursive function have overhead higher than programs that do not contain recursive function. We conclude from this, that this type of protection instructions are technically feasible and fast. As we discussed in our implementation in section 3, we add our protection instruction before each called function and in the end of each called function. Therefore, as the number of called function increases in the program, the accumulative execution time of program will increase too. To verify the protection level of our software tool, we tested it using known types of stack based attacks against return address.

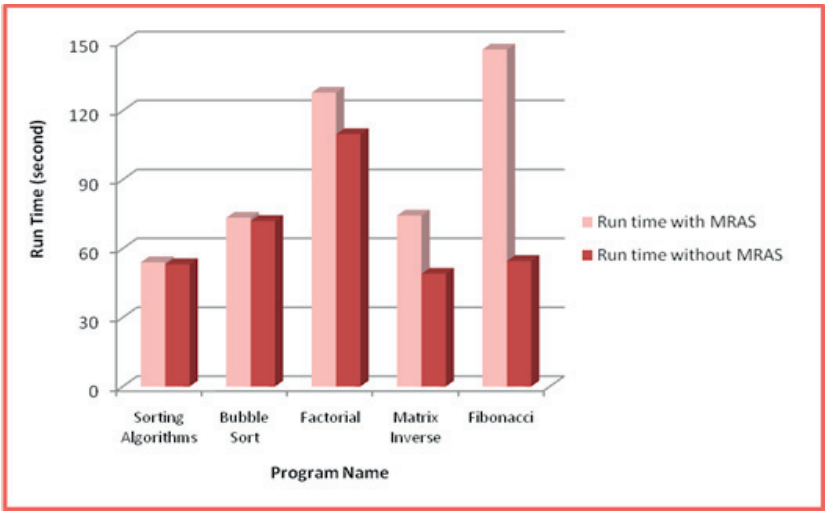

Fig. 5 Run time of macro programs with and without MRAS protection.

5. Conclusion

In this research, we proposed, implemented a general approach to achieve full protection for stack return addresses from all types of attacks. The main idea in our solution is to create a random number of copies of stack return addresses and store them at random locations. When a pointer is used, all copies are read and compared. In order for an attack to succeed, the attacker must know the number of copies, their locations, and be able to modify all of them simultaneously. This is an extremely difficult process. The proposed software tool was tested using several kinds of known micro benchmarks to verify its performance and the results show a small overhead comparing to reference time. Moreover, the software tool proved a high protection against integer overflow attack and buffer overflow attack.

References

[1]. Yong-Joon Park and Gyungho Lee, Repairing Return Address Stack for Buffer Overflow Protection 2004; ACM Frontiers of Computing, Italy.
[2]. J. Wilander, M. Kamkar, A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention 2003, in Proceedings of the 10th Network and Distributed System Security Symposium, pages 149--162, San Diego, CA.
[3]. M. Prasad and Tzi-cker Chiueh, A Binary Rewriting Defense against Stack based Buffer Overflow Attacks 2003; State University of New York at Stony Brook.
[4]. C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, Buffer overflows: Attacks and defenses for the vulnerability of the decade 2000; In Proceedings of the DARPA Information Survivability Conference and Expo (DISCEX), pages 119–129, Hilton Head, South Carolina.
[5]. CERT Computer Emergency Response Team statistics 2000-2008. http://www.cert.org/stats/vulnerability_remediation.html
[6]. Seon-Ho Park, Young-Ju Han, Soon-jwa Hong, Hyoung-Chun Kim, and Tai-Myoung Chung, The Dynamic Buffer Overflow Detection and Prevention Tool for Windows Executables Using Binary Rewriting 2007; Vol.3, International Conference on Advanced Communication Technology(ICACT).
[7]. Tzi-Cker Chiveh and Fu-Hau Hsu, RAD: A compile-time solution to Buffer Overflow Attacks 2001; Proceeding of 21st International conference on Distributed Computing system.
[8]. Vendicator. Stackshield, A stack smashing technique protection tool.
[9]. Francesco Gadaleta, Yves Younan, Bart Jacobs, Wouter Joosen, Erik De Neve, and Nils Beosier, Instruction-level countermeasures against stack-based buffer overflow attacks 2009; European Conference on Computer Systems, Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems.
[10]. Crispin Cowan, Calton Pu, David Maier, Heather Hinton, Peat Bake, Steve Beattie, Aron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Detection and prevention of Buffer-Overflow Attacks 1998; Proceeding of the 7th USENIX security symposium.
[11]. A. Tyagi, and G. Lee. Encoded program counter: Self Protection from Buffer Overflow Attacks 2000; Proceedings of International conference on Internet Computing (IC'2000).
[12]. C. Pyo and Gyungho Lee. Encoding Function Pointers and Memory Arrangement hecking against Buffer Overflow Attack 2002; Vol. 2513, Proceeding of the Fourth International Conference on Information and Communications Security, Singapore.
[13]. R.W.M. Jones and P.H.J. Kelly. Backward-compatible bounds checking for arrays and pointers in C programs 1997; Proceedings of the 3rd International Workshop on Automated Debugging.
[14]. A. Baratloo, N. Singh, and T. Tsai. Libsafe: Protecting critical elements of stacks December 1999.
[15]. Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks 2000; Proceedings of the USNIX Annual Technical Conference.
[16]. Solar Designer. Non-Executable user stack. http://www.openwall.com/.
[17]. Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending embedded systems against control flow attacks 2009; Conference on Computer and Communications Security, Proceedings of the first ACM workshop on Secure execution of untrusted code.
[18]. J. Xu, Z. Kalbarczyk, S. Patel, and R. Iyer. Architecture support for defending against buffer overflow attacks 2002.
[19]. S. Alexander. Defeating compiler-level buffer overflow protection 2005; Login; 30(3).
[20]. CASH: Checking Array Bound Violation Using Segmentation Hardware, www.ecsl.cs.sunysb.edu/softsecure/project.html.
[21]. F. Giasson. Memory layout in program execution 2001; http://www.decatomb.com/articles/memorylayout.txt.
[22]. Mark W. Eichin and Jon A.Rochlis. With microscope and tweezers: An analysis of the Internet virus of November 1988 1989;. Proceeding of the IEEE Symposium on Research in Security and Privacy