Control Flow Protection Against Return Oriented Attacks

Álvaro Rincón[1,2], Davidson Boccardo[2], Luci Pirmez[1], Luiz Fernando Rust[1,2]

[1]Programa de Pós-Graduação em Informática – Instituto Tércio Pacitti / Instituto de Matemática – Universidade Federal do Rio de Janeiro – 21.941-901 – Rio de Janeiro

[2]Instituto Nacional de Metrologia, Qualidade e Tecnologia - Av. N. S. das Graças, 50 – 25.250-020 – Xerém – Duque de Caxias – Rio de Janeiro

alvaro.rincon@ppgi.ufrj.br, luci@nce.ufrj.br

{drboccardo, lfrust}@inmetro.gov.br

Abstract. *Recent ROP (Return Oriented Programming) attacks are characterized by evading traditional protection methods, encouraging the scientific community to seek for a reliable and practical security solution. This work presents a novel technique based on control flow protection, and with a low overhead, making it suitable for constrained architectures in terms of processing, storage and energy. A prototype of the protection technique was developed and tested for ARM-Linux environment. The results show that our solution is effective and capable of preventing such ROP attacks with negligible overhead.*

1. Introduction

The amount of electronic devices with embedded software that makes part of the peoples' lives is growing daily due to the scientific advances and the lower cost of hardware components. As an example, we have smart meters doing telemetry to read the energy consumption of the end-users without human intervention [Huang et al. 2012]. The benefits are obvious, they increase reliability and efficiency of the processes. However, they also raise questions regarding the software security. The literature shows that software security flaws are growing rapidly [Alhazmi et al. 2007].

Recent studies have shown that it is possible to adapt techniques strictly designed to attack conventional systems to embedded systems. For example, in [Itzahak et al. 2011] the authors present an attack non-executable stacks, intrinsic characteristic of most embedded systems, allowing stack exploitation and control flow redirection. Code reuse techniques that use gadgets (snippets of the code with a control transfer instruction in the end) of the own software to manipulate the control-flow execution. These gadgets can be in any part of the software and are made up of a small set of instructions. The primary goal of these code instructions is to enable features that allow an unexpected software behavior, for instance, a control-flow redirection or a sensitive data modification. Such techniques have emerged in response to protection methods that distinguish between data and code instruction in memory, making the stack non-executable, thereby, preventing the execution of instructions within the stack.

Different types of techniques to protect systems against code reuse attacks have been developed in recent years [Abadi et al. 2009, Bletsch et al. 2011, Pappas et al. 2012] and can be broadly classified into two categories: static and dynamic. Static techniques aim at protecting anomalous control flows by instrumenting the source code

 ©2015 SBC — Soc. Bras. de Computação

before the compilation-time. The main limitation of the proposed static works is the priori knowledge of all execution paths of the software application. The dynamic techniques are based on code execution and dynamic binary instrumentation, foreseeing potential control flows' misdirection. In spite of the accuracy of the dynamic methods, the overhead is prohibitive for using in restricted environments, such as embedded devices that have very limited computing resources. Therefore, it is important to seek protection methods that are general, not needing the a priori knowledge of all execution control flow paths, and with a low overhead in terms of memory and processing consumption.

This paper proposes a static protection technique of the software control-flow against return-oriented attacks without needing the previous knowledge of all the execution paths and with a negligible overhead in the software application. It allows its use in restricted environments, like embedded systems, aiming to mitigate unauthorized control flow or data manipulation of the software application for return-oriented attacks. Broadly speaking, the technique inserts verification instructions at compile-time in order to protect gadgets that can be used maliciously in ROP attacks. These instructions have the ability to verify during runtime if the execution of the code instructions within of the basic blocks (group of sequential code instructions) is a legitimate control flow path. The legitimacy is determined when the execution flow starts at the first instruction and ends at the last instruction of each basic block.

2. Proposed approach

We propose a technique that differs between an authentic flow and a malicious flow of the software application, being the malicious performed by Return Oriented Programming (ROP). Our technique is based on the inspection of the execution flow of the basic blocks. Analogously, state inspection techniques [Christian et al. 2009] are those able to verify that the manipulation of a variable in the code is, in fact, legitimate, using verifiers. Similarly, the proposed technique also makes use of verifiers, but instead of checking a variable, the verifiers check whether the block execution flow started in its first instruction.

The basic blocks are obtained through the control flow graph (CFG) of the software application, obtained in the static analysis. A control flow graph is a representation of all execution paths that might be traversed during the software execution. Our technique uses the CFG to identify the basic blocks and the instructions contained therein. Thus, it is possible to identify and protect the basic blocks that contain snippets of code (gadgets) with the ability to generate ROP attacks.

Our technique instruments the basic blocks that contain gadgets through verifier instructions called *Assign* and *Checkpoint*. These verifiers will be located at specific parts within the basic blocks. The *Assign* instruction is inserted at the beginning the basic block and initializes a variable "x". At the end of the gadget, before the redirection flow instruction, the variable "x" is verified by the *Checkpoint* instruction. This verification determines if the current "x" value is the expected value for the code execution, that is, the value that was assigned at the beginning of the block, or if the execution flow is malicious, that is, the value of "x" differs form the value previously assigned. In the case of divergence, a ROP attack is detected and a response mechanism

 ©2015 SBC — Soc. Bras. de Computação

may be implemented, such as, stopping the application execution. Figure 1 shows a basic block without the protection technique (Figure 1a) and the same basic block with our protection technique (see Figure 1b).

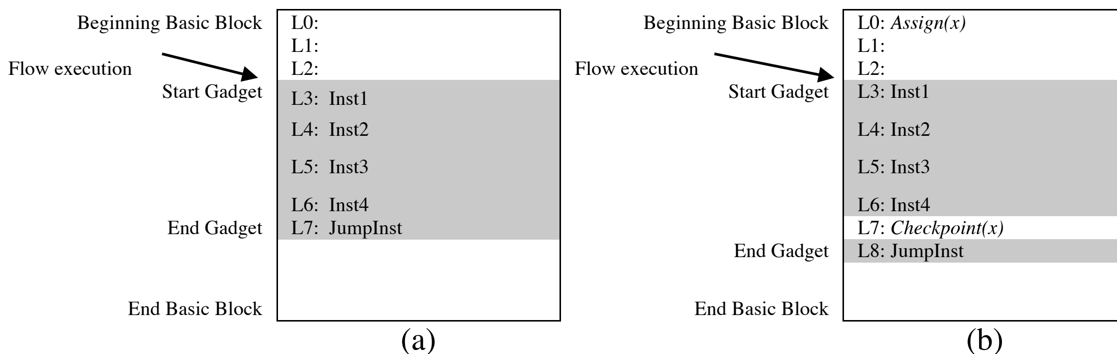


Figure 1. Basic block without protection (a) Basic block with protection (b).

The whole process of the proposed technique can be divided in three steps:

1) Assembly code and CFG generation: it uses compilation flags to obtain the assembly code with the symbol and relocation tables. The assembly code generated will be used to conduct the static analysis in order to obtain the CFG.

2) Gadgets identification and basic block instrumentation: it uses the assembly code to identify instruction sets that can be used as gadgets in a ROP attack. These instruction sets must have some specific characteristics, such as ability of manipulate registers used to pass arguments to a function and have a jump instruction or return instruction at their ends. After identification of the instruction sets, is necessary to identify the blocks in which they are contained to insert the verifier instructions *Assign* and *Checkpoint*.

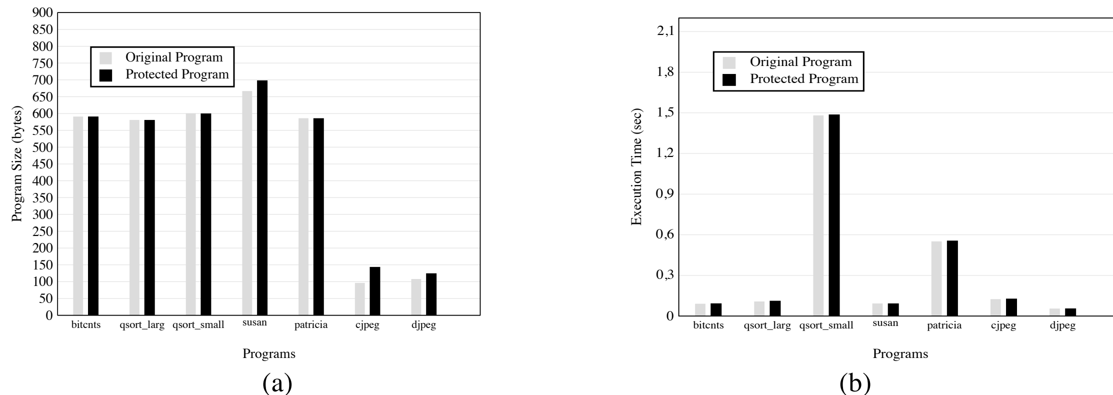
3) Protected executable code generation: It comprises the compilation of the instrumented assembly code, generating the protected executable code.

3. Experiments

This section describes as our technique to protect basic blocks against ROP attacks was validated. The experiments were developed in a processor ARMv5TEJ in an environment ARM-Linux with kernel 3.2.0-4-versatile virtualized through the QEMU tool. The tools used to perform the protection were: compiler GCC 4.6.3, assembler AS 2.22 and debugger GDB 7.4.1. To perform the validation has been taken into account the impact generated in terms of (i) code size and (ii) computational cost. The metrics used, respectively, are: (i) code size in bytes and (ii) execution time in seconds.

Figure 2.a shows the difference between the size in bytes before and after applying protection. Therein is possible to see that the greater the size of the executable less overhead in terms of bytes, that means, that our protection technique is inversely proportional to code size. For instance, the *bitcnts* executable overhead with 587,628 bytes is 0.01% and *cjpeg* with 93,056 bytes is 50.92%. Figure 2.b shows the execution times of the executables with and without protection. Therein is possible to see the average of our protection technique causes an overhead in order of 2.3% with a standard deviation of 1.7%. The times were obtained through routine time() executed 20 times for each executable.

 ©2015 SBC — Soc. Bras. de Computação



(a) (b)

Figure 2. Experiments about program size (a)and time execution (b).

4. Conclusions

This paper presents a technique to protect basic blocks using verifiers with the ability to detect a ROP attack when the execution of the basic blocks is not performed in a conventional way, that is, the execution flow does not start from the first instruction of the basic block. The results show that the technique has minimal overhead in terms of time and space, proving its suitability to protect restricted systems, such as embedded systems. Additionally, according to our research, this is the first static protection technique against ROP attacks to ARM architecture.

References

Abadi, Martín, et al. (2009) "Control-flow integrity principles, implementations, and applications." *ACM Transactions on Information and System Security (TISSEC)* 13.1 (2009): 4.

Alhazmi et al. 2007 Alhazmi, O. H., Malaiya, Y. K., and Ray, I. (2007). Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers & Security*, 26(3):219–228.

Bletsch, Tyler, Xuxian Jiang, and Vince Freeh. (2011) "Mitigating code-reuse attacks with control-flow locking." *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM.

Christian Collberg and Jasvir Nagra. (2009). *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection* (1st ed.). Addison-Wesley Professional.

Huang, Zi-Shun, and Ian G. Harris. (2012) "Return-oriented vulnerabilities in ARM executables." *Homeland Security (HST), 2012 IEEE Conference on Technologies for*. IEEE.

Itzhak(Zuk) Avraham. (2011) Non-Executable Stack ARM Exploitation Research Paper. In BlackHat Security Convention. https://media.blackhat.com/bh-dc-11/Avraham/BlackHat-DC-2011-Avraham-ARMExploitation-wp.2.0.pdf

Pappas, Vasilis, Michalis Polychronakis, and Angelos D. Keromytis. (2012) "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization." *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE.

 ©2015 SBC — Soc. Bras. de Computação