SAJET

South Asian Journal of Engineering and Technology Vol.2, No.15 (2016) 87–92

ISSN No: 2454-9614

Detection of Code Reuse Attack Using Code Randomization and Data Corruption Approach

N. Mohanappriya[a], R. Rajagopal[a*]

[a)] *Department of Computer Science and Engineering, Vivekanandha Institute of Engineering and Technology for Women, Tiruchengode , Tamilnadu, India.*
[b)] *Department of Computer Science and Engineering, Vivekanandha Institute of Engineering and Technology for Women, Tiruchengode , Tamilnadu, India.*

*Corresponding Author: N.Mohanappriya
E-mail: mohanapriyacse29@gmail.com,

Received: 15/11/2015, Revised: 20/12/2015 and Accepted: 25/02/2016

Abstract

Code-reuse attacks, such as return-oriented programming (ROP), are a class of buffer overflow attacks that repurpose existing executable code towards malicious purposes. These attacks bypass defences against code injection attacks by chaining together sequence of instructions, commonly known as gadgets, to execute the desired attack logic. A common feature of these attacks is the reliance on the knowledge of memory layout of the executable code. A fine grained randomization based approach that breaks these assumptions by modifying the layout of the executable code and hinders code-reuse attack. Our solution, Marlin, randomizes the internal structure of the executable code by randomly shuffling the function blocks in the target binary. This denies the attacker the necessary a priori knowledge of instruction addresses for onstructing the desired exploit payload. Our approach can be applied to any ELF binary and every execution of this binary uses a different randomization. The integrated Marlin into the bash shells that and omizes the target executable before launching it. Our work shows that such an approach incurs low overhead and significantly increases the level of security against code-reuse based attacks.

Keywords: Return oriented programming, code randomization, security, malware.

Reviewed by* **ICETSET'16 *organizing committee*

1. Introduction

Return Oriented Programming (ROP) attacks are an advanced form of buffer overflow attacks **[1]** that reuse existing executable code towards malicious purposes. While earlier exploits involved the injection of malicious code **[1],** the recent trend has been to reuse executable code that already exists, primarily in the application binary and shared libraries such as libc. These code reuse attacks can bypass traditional defences against code injection attacks such asW _ X protection **[2]**that prevents execution of arbitrary code that is injected into the memory. In a basic code reuse attack, for instance return-into-libc attack **[3],** a buffer overflow corrupts the return address to jump to a libc function, such as system. This type of attack then evolved into a more generic ROP attack **[4]**. In ROP, the

attacker identifies small sequences of binary instructions, called gadgets, that end in a ret instruction. By placing a sequence of carefully crafted return addresses on the stack, the attacker can use these gadgets to perform arbitrary computation. These attacks continued to evolve, with newer techniques using gadgets that end in jmp or call instructions **[5]**. As these attacks rely on knowing the location of code in the executable and libraries, the intuitive solution is to randomize process memory images. In basic address space layout andomization (ASLR), the start address of the code segment is randomized. First, the main shortcoming of earlier randomization-based techniques was insufficient entropy, thus making brute-force attacks feasible. Second, executable code can naturally be broken into many function blocks that can potentially be shuffled. Consequently, the amount of possible randomization generated can be significantly increased by permuting these code blocks within the executable. For instance, if an application has 500 function blocks, there are 500! _ 23; 767 possible permutations of these function blocks which significantly increases the brute force effort required from an attacker. Our system, Marlin, introduces a randomization technique that shuffles the function blocks in an application binary. This technique is integrated into a customized bash shell that randomizes the target binary at load time just before execution. This randomization approach has many benefits. First, as stated above, for any considerably sized code base with a large number of function blocks, the number of possible randomized results clearly makes brute-force attacks infeasible. Improved our earlier implementation of marlin to make the binary randomization much faster. Finally, in the current paper we include a more extensive set of experiments to evaluate the Marlin technique.

2. Marlin Defence Technique

Code-reuse attacks make certain assumptions about the address layout of application's executable code. Marlin's randomization technique aims at breaking these assumptions by shuffling the code blocks in the binary's .text section with every execution of this binary. This significantly increases the difficulty of such attacks since the attacker would need to guess the exact permutation being used in the current process execution. This shuffling is performed at the granularity of function blocks. The various steps involved in Marlin processing. Marlin is integrated into a modified bash shell that randomizes the target application just before the control is passed over to this application for execution.

3. Randomization Algorithm

The randomization algorithm described in Algorithm 1 involves two stages. In the first stage, the function blocks are shuffled according to a certain random permutation. During this shuffling, we keep a record of the original address of the function and also the new address where the function will reside after the binary has been completely randomized. This information is stored in a jump patching table. Note that this jump patching table is discarded before the application is given control, thus preventing attacker from utilizing this information to de-randomize the memory layout. In the second stage, the actual jump patching is executed where the jump patching

table is examined for every jump that needs to be patched. Whenever a relative jump is encountered, the algorithm executes the PatchRelativeJump() method to redirect the jump to the correct address in the binary. The PatchRelativeJump() method takes the current address of the jump and the address of the jump destination to determine the new offset and patch the jump target. The second case is the computed jumps where the contents of a register specify the absolute address of the destination, for example call to function pointers. We handle these cases by doing a backward analysis and fixing the instruction where the function address is being loaded into a register. If the destination address is obtained from .data section (for example, in case of global function pointers), then we patch the .data section with the new value. This processing is done by the PatchAbsoluteJump() method. Thus, to defeat Marlin, an attacker would need to dynamically construct a new exploit for every instance of every application which is not possible since the randomized layout is not accessible to the attacker. We now discuss the security guarantees offered by Marlin.

Algorithm 1. Code Randomization algorithm

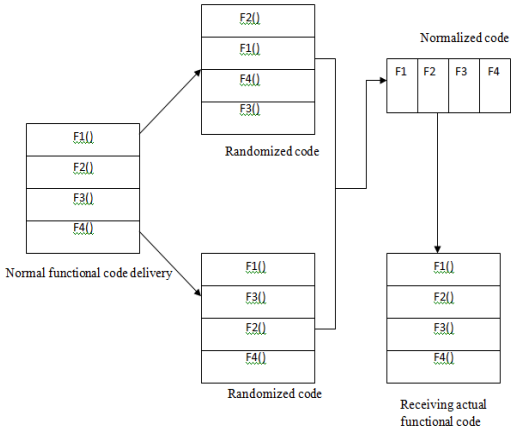
Input: Original program, P
Output: Randomized program, P_R
L = All symbols in P
F = A list of forbidden symbols that should not be shuffled
$L = L - F$
O_L = Ordered sequence of symbols in L
$S.Addr_P$ = Address of symbol S in program P
$J.Addr_P$ = Address of jump instruction J in program P
$J.Dest_P$ = Destination address of jump J in program P
$J.Sym$ = Symbol that J is jumping into

```
/* Permutation stage */
```
for *Every symbol* $S \in L$ **do**
    R = Randomly select another symbol in L
    Swap S and R in O_L
P_R = Permuted program according to symbol order in O_L

```
/* Jump patching stage */
```
for *Every symbol* $S \in L$ **do**
    for *Every jump* $J \in S$ **do**
        if *J is a relative jump to within S* **then**
            /* No action needed */
        else if *J is a relative jump to outside S* **then**
            $J.Dest_{P_R} = J.Dest_P + (J.Sym.Addr_{P_R} - J.Sym.Addr_P) - (S.Addr_{P_R} - S.Addr_P)$
            PatchRelativeJump($J.Addr_{P_R}$, $J.Dest_{P_R}$)
        else if *J is an absolute jump* **then**
            PatchAbsoluteJump($J.Addr_{P_R}$, $J.Dest_{P_R}$)

4. Randomization Stage

In this stage, the actual shuffling of the function blocks is performed. The first step is to generate a random permutation of symbols and shuffle the list of symbols to obtain a new order of symbols. The new binary is re-written according to this new symbol order. In our preliminary implementation **[8]**, we did not shuffle certain symbols such as _start that were referred to as forbidden symbols. Our revised implementation no longer has this limitation and all the symbols within .text section are now randomized, including _start symbol. This _start symbol is the first instruction that executes after the binary is loaded into the memory by the ELF loader. This entry address is stored in ELF header of the binary. Once the application is randomized, we patch the ELF header with the new entry address which is the new location of _start symbol.



5. Optimization Techniques

A straightforward performance optimization for Marlin would be to perform the pre-processing for jump patching only once for each application and store the result in a database maintained by the system. The jump patching algorithm can reuse the information about function blocks from this database in subsequent executions. The database would only need to be updated when the application code changes. The impact of the code randomization can be reduced by taking the permutation generation off-line. To do so, each application will have a dedicated file containing the next instance's permutation. When a binary is executed, the custom shell sends a signal to a trusted daemon process that runs with low priority and returns the next permutation. The application's function blocks are then shuffled accordingly.

5.1 File Deploying

In this module, we are going to connect the network .Each node is connected the neighbouring node and it is independently deployed in network area. And also deploy the each port no is authorized in a node. The intrusion detection is defined as a mechanism for a WSN to detect the existence of inappropriate, incorrect, or anomalous moving attackers. In this module check whether the path is authorized or unauthorized. If path is authorized the

packet is send to valid destination. Otherwise the packet will be deleted. According port not only we are going to find the path is authorized or Unauthorized.

5.2 Code Randomization

Randomizing an application's executable code segment consists of two stages. First is the pre-processing stage that can be done just once per binary and is independent of subsequent executions. This stage involves disassembling binary and extracting information about the function blocks and also the control flow. The second stage is the actual randomization stage when the function blocks are shuffled and the jump/call targets are patched. We now discuss each of this in further detail..We achieve this by performing jump patching. Fine-grained address space layout randomization (ASLR) has recently been proposed as a method of efficiently mitigating runtime attacks. In this paper, we introduce the design and implementation of a framework based on a novel attack strategy, dubbed just-in-time code reuse, which undermines the benefits of fine-grained ASLR. Specifically, we derail the assumptions embodied in fine-grained ASLR by exploiting the ability to repeatedly abuse a memory disclosure to map an application's memory layout on-the-fly, dynamically discover API functions and gadgets, and JIT-compile a target program using those gadgets—all within a script environment at the time an exploit is launched. We demonstrate the power of our framework by using it in conjunction with a real-world exploit against Internet Explorer, and also provide extensive evaluations that demonstrate the practicality of just-in-time code reuse attacks. Our findings suggest that fine-grained ASLR may not be as promising as first thought.

5.3 Code Reuse Attack

The attacker identifies small sequences of binary instructions, called gadgets, that end in a ret instruction. By placing a sequence of carefully crafted return addresses on the stack, the attacker can use these gadgets to perform arbitrary computation. These attacks continued to evolve, with newer techniques using gadgets that end in jmp or call instructions. As these attacks rely on knowing the location of code in the executable and libraries, the intuitive solution is to randomize process memory images. In basic address space layout randomization (ASLR), the start address of the code segment is randomized. That is, two different running instances would have a different base address, so the addresses that an attacker needed to jump to in one instance would not be the same as the addresses in the other instance. Although said approach initially seemed promising, 32-bit machines provide insufficient entropy as there are only 216 possible starting addresses. This makes the approach vulnerable to brute-force attacks.

5.4 Code Normalization

The relative offsets of instructions within the application's code are constant. That is, if an attacker knows any symbol's address in the application code, then the location of all gadgets and symbols in applications codebase is deterministic. Protection against traditional ROP: The ROP adversary analyzes the code and constructs the gadget chain prior to the execution of the targeted (vulnerable) application. Hence, we require a mechanism that changes the addresses of the gadgets and consequently breaks the gadget chain. As described in Section II-A, any (fine-grained) randomization, applied before the application is executed, is suitable.

6. Conclusion and Future work

To defend against code-reuse attacks was to increase the entropy by randomizing the function blocks. One may apply this randomization technique at various levels of granularity—function level, block level or gadget level. The level of granularity to choose is a trade off between security and performance. In our implementation, we implemented the randomization at the function level which is the most coarse granularity amongst the three mentioned above. However, we show that even this coarse level of granularity provides substantial randomization to make brute force attacks infeasible. Our prototype implementation requires the binary disassembly to contain symbol information, i.e. a non-stripped binary. In practice however, binaries may be stripped and not contain the symbol information. We address this by using external tools such as Outstrip **[39]** that restore symbol information to a stripped binary. Another approach to process stripped binaries is to randomize at the level of basic blocks since they do not require function symbols to be identified. However, randomizing at basic block granularity will likely incur higher runtime overhead as it would break the principle of locality. One limitation of Marlin is that it is unable to correctly rewrite certain binaries if these target binaries have certain compiler optimizations enabled or if they are obfuscated. This is because Marlin requires the: text section in the target binary to be organized as function blocks and for these function block to be clearly identifiable using disassemble. In this work, we proposed a fine-grained randomization based approach to defend against code reuse attacks. This approach randomizes the application binary with a different randomization for every run.

References

[1] Aleph One, “Smashing the stack for fun and profit,” Phrack Mag., vol. 49, no. 14, Nov. 1996.
[2] PaX Team. PaX [Online]. Available: http://pax.grsecurity. Net/(2003).
[3] Solar Designer, “Getting around non-executable stack (and fix),” Aug. 1http://seclists.org/bugtraq/1997/Aug/63.
[4] H. Shacham, “The geometry of innocent flesh on the bone: Returninto- libc without function calls (on the x86),” in Proc. 14th ACM Conf. Comput. Commun. Security, 2007, pp. 552–561.
[5] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, “Return-oriented programming without returns,” in Proc. 17th ACM Conf. Comput. Commun. Security, 2010, pp. 559–572.
[6] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, “On the effectiveness of address-space randomization,” in Proc. 11th ACM Conf. Comput. Commun. Security, 2004, pp. 298–307.
[7] G. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, “Surgically returning to randomized lib(c),” in Proc. Annu. Comput. Security Appl. Conf., Dec. 2009, pp. 60–69.
[8] A. Gupta, S. Kerr, M. Kirkpatrick, and E. Bertino, “Marlin: A fine grained randomization approach to defend against ROP attacks,” in Proc. 7th iNt. Conf. Netw. Syst. Security, 2013, vol. 7873, pp. 293–306.
[9] A. Gupta, S. Kerr, M. S. Kirkpatrick, and E. Bertino, “Marlin: Making it harder to fish for gadgets,” in Proc. ACM Conf. Comp. Commun. Security, 2012, pp. 1016–1018.
[10] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, “When good instructions go bad: Generalizing return-oriented programming to RISC,” in Proc. 15th ACM Conf. Comput. Commun. Security, 2008, pp. 27–38.
[11] R. Hund, T. Holz, and F. C. Freiling, “Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms,” in Proc. 18th Conf. USENIX Security Symp., 2009, pp. 383–398.
[12] A. Francillon and C. Castelluccia, “Code injection attacks on harvard-architecture devices,” in Proc. 15th ACM Conf. Comput. Commun. Security, 2008, pp. 15–26.
[13] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, “Privilege escalation attacks on android,” in Proc. 13th Int. Conf. Inf. Security, 2011, pp. 346–360.