Reduction of Code Reuse Attacks Using Code Randomization and Recursive Traversal Algorithm

K. Krishna priya and Dr.P.Murugeswari

1PG scholar, Department of CSE, Sri Vidya College of Engineering & Technology, Virudhunagar, India.
2Head of the Department, Department of CSE, Sri Vidya College of Engineering & Technology, Virudhunagar, India.

Received February 2016; Accepted 18 April 2016; Available 25 April 2016

ABSTRACT

Code-reuse attacks, including return-oriented programming (ROP), are a class of buffer overflow attacks that shows the existence of executable code that can be used for malevolent purposes. These attacks skip defenses towards code injection attacks by using chaining together collection of instructions, commonly referred to as gadgets, to execute the desired attack logic. A common characteristic of these attacks is the reliance at the understanding of memory format of the executable code. A fine grained randomization based approach that breaks those assumptions with the aid of modifying the format of the executable code and hinders code-reuse attack. Marlin technique change the internal structure of executable code by shuffling the target binary's function blocks in random manner. This refuse the attacker a priori information of instruction addresses for constructing the preferred exploit payload. This approach can be carried out to any ELF binary and each execution of this binary makes use of an extraordinary randomization. Marlin can be incorporated into the bash shell that randomizes the target executable before launching it. The shuffled function blocks of the applications binary are executed through the compiler and at some time it will be unable to rewrite certain binaries if they are obfuscated. This confusion will be avoided by the Recursive traversal algorithm implemented in sequencer. Thus this method reduces the vulnerability of security against attacks based on code reuse.

KEYWORDS: Code reuse attacks, return oriented programming, Randomization, Malware, sequencer

INTRODUCTION

Return oriented programming (ROP) attacks are a superior form of buffer overflow assaults [2] that reuse existing executable code towards malevolent purpose. In ROP, the attacker identifies small sequences of binary instructions, called gadgets, that lead to a ret preparation. By way of setting a chain of vigilantly crafted go back addresses at the stack, the attacker can use these gadgets to carry out arbitrary computation. These attacks persevered to evolve, with more recent strategies the usage of gadgets that end in jmp or call instructions [3].

As those assaults rely on understanding the vicinity of code in the executable and libraries, the intuitive solution is to randomize method memory images. In basic address space layout randomization (ASLR), the begin cope with of the code segment is randomized. That is, two dissimilar running instances might have a unique base cope with, so the addresses that an attacker needed to leap to in one example might no longer be the same as the addresses within the different instance. Even though stated technique first of all appeared promising, 32-bit machines offer insufficient entropy as there are handiest 216 feasible starting addresses. This makes the technique vulnerable to brute-force attacks [4].

Our approach is to revisit the granularity at which randomization is completed. As opposed to randomizing handiest a single parameter, our approach (Marlin) breaks an application binary into blocks of code and shuffles them. This substantially will increase the entropy of the system as an example, and software with 500 code
blocks allows for $500! \approx 2.3767$ variations, making brute-force attacks infeasible. Our method, which can be carried out to any ELF binary without requiring supply code, is accomplished transparently at load time to ensure every execution example is particular. Ultimately by means of paying a performance price up the front, Marlin avoids the overhead of on-going monitoring of essential statistics, such as go back addresses, which other structures impose.

**Related Work:**

This section presents the brief summary of ROP attacks and existing defenses.

**A. Return-Oriented Programming:**

ROP takes advantage of an approach that has developed from stack-based buffer overflows. In ROP exploits, an attacker crafts a series of gadgets which are present in existing code to perform arbitrary computation. A gadget is a small collection of binary code that results in a ret instruction. By vigilantly crafting a chain of addresses on the software program stack, an attacker can manipulate the ret instruction semantics to leap to arbitrary addresses that correspond to the start of gadgets. Doing so allows the attacker to carry out arbitrary computation. ROP strategies may be used to create rootkits, can inject code into Harvard architectures, and had been used to perform privilege escalation in Android. The identical method of stringing together gadgets has been used to govern other instructions, inclusive of jmp and their editions [16], [18].

**B. Defenses:**

The two well-known techniques for defending against code-reuse attacks are Address space layout randomization (ASLR) and address obfuscation [17]. Address obfuscation and ASLR both have the same inadequacy for randomizing the instruction set in that the small amount of randomization leaves applications vulnerable to attacks [4].

Another approach for code-reuse attacks is to come across and terminate the attack when it occurs. DROP [5] is a binary display applied method; it detects ret commands and initiates a dynamic evaluation routine, primarily based on a statistical analysis of customary application behavior. While a ret instruction would end in a handle with in libc, DROP decides if the current execution routine exceeds a candidate gadget length threshold. These thresholds are primarily based on a static evaluation of ordinary application behavior. The binary must be compiled with DROP enabled. Another approach, DynIMA [6] framework is the finest framework that gives both load-time and runtime integrity for program binaries without information of their source code and even below the presence of attacks which are based on return-oriented programming.

Other methods store sensitive data, inclusive of return addresses, on a shadow stack and confirm their integrity before use. L.Davi et.al [7] suggests a method, ROPdefender which detects the conventional ROP attacks which can be based on return commands. The ROPdefender inspects the instruction type, at some stage in the execution and then identify the return address destructions to prevent ROP attacks. It uses instrumentation to analyze ROP attacks which can be carried out at during runtime or at compile time. Defense techniques that monitor and/or lock control flow [8], [9] to stop ROP attacks have also been proposed. The technique called as CFL [8] ensures that the control flow graph of an application is deviated from no more than once, and that this deviation cannot be used to craft a malevolent system call. CFL works by performing a “lock” operation before each indirect control flow transfer, with a matching “unlock” operation present at valid destinations only. To mitigate code-reuse attacks CFL presents a step further with a small overall performance penalty. C.Zhang et.al [9] suggests a new approach called CCFIR to ensure that indirect control transfers jump only to known targets. It can be used to enforce CFI, which provides a solid base for software protection. It can block various attacks against control transfers, including most ROP attacks. CCFIR can be applied through binary rewriting on executables generated by modern compilers. The downside of this approach is there is a chance to alter pointers that flow to external modules which might be unprotected for the reason that CCFIR is applied only to parts of program.

Some recent research approaches also explore the idea of software diversity as a defense against ROP attacks. ILR [10] randomizes location of every instruction in the application code and thus preventing the attacker to reuse program functionality. They function on arbitrary executable applications and don’t require compiler support with no user interaction. Post deployment ILR can be automatically functional and alleviates recurrent re-randomization and it is based on a process-level virtual system that incurs an overall performance price during the period of the application. In contrast, Marlin’s performance impact is primarily limited to the start-up cost.

Gadget chain detection method [11] suggests ROPecker technique, which defends all types of ROP attacks which do not need access to the source code. This method uses the gadget chain detection algorithm to detect the chain in execution flow and then the sliding window mechanism that triggers the algorithm in proper time.

Pappas et al. [12] suggests an in-place code randomization approach that probabilistically breaks 80 percent of the instruction sequences that are helpful for attacks. This method provides on narrow scope modifications in
code segments of executable code by means of the use of an array of code transformation strategies. Also the randomization of stripped binaries without complete disassembly coverage would be enabled to evade damage to the semantics. The purpose of this randomization process is to eradicate or probabilistically modify any number of gadgets. ROP code relies an accurate execution of all chained gadgets, if there is alteration in a few may result in ineffective ROP code. Also, Marlin randomizes the executable with every run unlike [12] and [13] that don’t re-diversify the binary. XIFER [14] and STIR [15] observe software diversification to an application at runtime to guard against code reuse attacks. The binary rewriting can be carried out at the load time of the application then it disassembles binary utility and performs code changes and assembles new application instances with new memory layouts. The randomization will take place dynamically thus it doesn’t use an offline static analysis. They’re no longer open to disclosure attacks as the diversification is carried out against for each application run. XIFER provides memory overhead, as the possibility to write out ELF executable or shared library file will increase the file size. R. Wartell et al. [15] proposed a completely automated and binary centric method STIR. This method does not need any source code or symbolic information for target binary program. Each time STIR is launched the code reorders the primary blocks in every binary code section randomly, hence disallowing the attempts for predicting the place of the gadgets. While XIFER and STIR apply diversification at the granularity of fundamental basic blocks, Marlin randomize at function block level and show that is enough to make brute force attacks infeasible. Additionally, these techniques could incur greater overhead than Marlin as they randomize at a very fine granularity. Marlin is a novel solution for thwarting ROP attacks that does not have the restrictions discussed above. A tremendous deal of software is distributed within the form of executable code. Such code is potentially susceptible to reverse engineering, in the form of disassembly observed by means of decompilation. This can permit an attacker to discover vulnerabilities inside the software, adjust it in unauthorized methods, or steal intellectual property via software program piracy. This paper describes and evaluates strategies that recursive traversal, are utilized by a number of binary translation and optimization structures [19].

C. Threat Model:

The proposed protection, Marlin, is aimed to defend a vulnerable application towards code reuse attacks, which include ROP attacks. This appliance may additionally have a buffer overflow vulnerability that may be leveraged through an attacker to inject an exploit payload. The device is thought to be protected by the use of W ⊕ X policy and the attacker can’t insert arbitrary executable code in the stack or the heap. The attacker is assumed to be aware of the functionality of Marlin. However, the attacker cannot examine the memory sell off of the running system and is unaware of how precisely the current executing process image of the code is randomized. This approach protects in the direction of far flung and local exploits as long as the attacker is not able to check the memory of the target system.

Defense Technique:

Code-reuse attacks craft certain assumptions concerning the address format of application’s executable code. Marlin’s randomization approach intends at breaking these assumptions by shuffling the code blocks within the .text section of binary and with every execution of this binary. This considerably increases the difficulty of such attacks for the reason the attacker would need to guess the exact permutation being used in the current process execution. This shuffling is done at the granularity of function blocks. Marlin is incorporated into a modified bash shell that randomizes the target application just before the control is passed over to this application for execution. Therefore, every execution of the program results in a distinctive process memory image.

D. Preprocessing Phase:

As cited above, Marlin randomizes the application binary on the granularity of function blocks. This requires figuring out the function blocks within the application binary. Preprocessing phase parses the ELF binary to extract the function symbols and associated data which includes start address of the function and length of the function block. However, traditional binary are normally stripped binaries and doesn’t hold symbol information. In such instances, first restore the symbol information using an outside tool, Unstrip [17]. Formerly the symbol information is restored and recognized, and then the next stage of Marlin processing that randomizes the application binary.

E. Randomization Algorithm:

Once the function symbols had been diagnosed, Marlin generates a random permutation of this set of symbols. The resulting permutation determines the order in which the memory map system calls are issued, which modifies the order of the mapped symbols in memory. The function blocks are then shuffled according to random permutation. Shuffling the code blocks in an application binary changes the relative offsets among instructions that may affect various jump instructions and those jumps may be either absolute jumps or relative
jumps. Relative jumps increment or decrement the program counter by a constant value and the absolute jump that directly jump to a fixed address. When randomizing the code blocks these jumps point the fixed locations and it can be done by the jump patching.

Aditi Gupta [1] et.al proposed the randomization algorithm can be explained in two phases. In the first phase, the function blocks are shuffled according to a certain random permutation. While shuffling the blocks, padding is added to ensure the resulting binary is page aligned. During this shuffling, preserve the record of the original address of the function and also the new address while the function will exist in after the binary has been completely randomized. This information is saved in a jump table and this table is discarded before the application is executed, therefore preventing the attacker from utilizing this information to derandomize the memory layout.

In the second phase, the actual jump patching is carried out and the algorithm examines the jump table for every jump that needs to be patched. The algorithm executes Patchrelativejump() method to forward the jump to the correct address in the binary while a relative jump is encountered. This method takes the current address of the jump and the address of the jump destination to decide the new offset and patch the jump target. The absolute jump method performs the computed jumps where the contents of a register specify the absolute address of the destination and this processing is done by PatchAbsoluteJump() method.

The run-time shuffling of the function blocks guarantee that multiple instances of the same program have unique address layouts. Therefore, to defeat Marlin, an attacker might dynamically construct a new exploit for every application which isn't always possible since the randomized layout isn’t accessible to the attacker.

F. Recursive Traversal Algorithm:

Recursive traversal algorithm determines the next instruction to disassemble via the control flow of the system. Whilst encountered the control transfer instruction, it will not keep disassembling sequentially, but begins to disassemble from the target address the control transfer instruction. The non control transfer instruction permits the disassemble, to skip over record blended into code sections. A distinctive feature of this algorithm is that, by following the control flow behavior of the program being processed, it is able to turn and consequently keep away from disassembly of data embedded inside the text section.

System Design:

A Marlin prototype can function on any ELF binary without requiring its source code. Implementation of Marlin involved two leading mechanisms. First part consisted of randomizing the executable code and generating the randomized binary then the second part treated integrating this into an existing system such that binary randomization occurs seamlessly with each execution.

G. Code Randomization:

Randomizing an application’s executable code section consists of two levels. First is the preprocessing stage that can be carried out once per binary and is impartial of subsequent executions. This stage includes disassembling a binary and extracting data about the function blocks and additionally the control flow. The second level is the actual randomization stage when the function blocks are shuffled and the jump/call targets are patched.

1. Preprocessing Stage:

The binary is randomized before the function blocks are identified. This could be done by disassembling the binary then parsing the dissembler output to extract the function symbols and the relevant records. For every function symbol accumulate information about its location in the executable, the length of the function block and the information on jumps or calls originating from this function.

2. Randomization Stage:

In this level, the actual shuffling of the function blocks is carried out. 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. Once the application is randomized, patch the ELF header with the new entry address which is the new area of _start symbol.
3. **Fixing Jumps and Calls:**
   The jump and call patching is accomplished in the identical pass while the new randomized binary is written. That is completed by using the patch list data that is generated in the preprocessing stage. For every call that requires to be patched, the patch information consists of the name of the first symbol, the name of symbol being patched to and the offset from the start of the parent symbol where the patching requires to be accomplished.

4. **Intimating Function Addresses to Compiler:**
   The function blocks are shuffled and then it will be executed through the sequencer. It uses the Recursive traversal algorithm, to intimate the shuffled binary address to compiler and it avoids the confusion of the compiler when executing the control transfer instructions.

**Security Evaluation:**
Fig.2 shows the randomization approach extensively increases the brute force effort required to attack the gadget. In a brute force attack, the attacker will randomly count on a reminiscence layout and craft exploit payload according with address layout. A failed attempt will typically reason a segmentation fault due to illegal guidance and the crashed technique or thread will need to be restarted.
Fig. 2: Marlin performance

Fig.3 shows the recursive traversal algorithm is used to infeasible the brute force attacks. The measured runtime overhead of randomized binaries affects the execution time of a binary. The execution time of un-randomized benchmarks as a baseline to evaluate with randomized benchmarks. Performed 40 randomizations per benchmark and took the average of those values the execution time of the benchmarks was identical earlier than and after randomization and became not affected because of recursive traversal method. This supports our initial hypothesis that the overhead is incurred only during the randomization phase of recursive traversal process and after that the binary executes as a normal binary with no runtime overhead. This algorithm supports our preliminary hypothesis that the binary executes as a normal binary without a runtime overhead.

Fig. 3: Recursive Traversal Performance

Conclusion:

In this paper we consider a solution, for defending code reuse attacks. We can achieve this through code randomization of function blocks. Function level randomization is the coarse level granularity. This technique randomizes the binary code and provides different randomization for every execution of the binary code. The shuffled function blocks of the applications binary are executed through the compiler and at some time it will be unable to rewrite certain binaries if they are obfuscated. This confusion will be avoided by the Recursive traversal algorithm implemented in sequencer. Thus this method reduces the vulnerability of security against attacks based on code reuse.

REFERENCES


