International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395 -0056
Volume: 02 Issue: 08 | Nov-2015 www.irjet.net p-ISSN: 2395-0072

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

K. Krishna priya[1], Dr.P.Murugeswari[2]

[1] *PG scholar, Department of CSE, Sri Vidya College of Engineering & Technology, Virudhunagar, Tamilnadu , India*

[2] *Professor, Department of CSE, Sri Vidya College of Engineering & Technology, Virudhunagar, Tamilnadu, India*

Abstract – ***Return oriented programming (ROP) and other code reuse attacks are a class of buffer overflow attacks that shows the existence of executable code that can be used for malicious purposes. They attack the systems security by chaining the sequence of instructions together to perform the expected logic of attack. These attacks have a common feature; they rely on executable code's memory layout. The layout of the executable code can be modified to avoid code reuse attacks. In marlin we change the internal structure of executable code by shuffling the target binary's function blocks in random manner. This will not allow the attacker to gain information of the instruction addresses, which will result in reduced possibility of attacks. Marlin can be implemented with any ELF binary code and every execution of the binary code will be using different randomization techniques. The target executable binary will be randomized before launching by integrating marlin to the bash shell. Thus our system reduces the vulnerability of security against attacks based on code reuse.***

Key Words: *Code reuse attacks, return oriented programming, code randomization*

1. INTRODUCTION

Network security describes the policies and procedures that are implemented through a network administrator, for avoiding and tracking unauthorized access or usage of network and its resources. If implemented properly network security will block malware, viruses and hackers from accessing the information in the system. Network security's first layer usually demands a username and password, thus allowing authorized users with certain privileges. Once the user is authenticated with certain permissions to access the system, the firewall enables network policies for the user, but they cannot detect malware or viruses. So an intrusion prevention system or antivirus is used for screening the user's access. In network security the main policy is to protect the assets, i.e. information, user accounts, passwords, server configurations etc. An attack in the system can be of two types, the information can be monitored which is passive attack and the information will be altered/destroyed or the whole network can be corrupted which is active attack. Without proper security the network can be attacked in any way.

2. LITERATURE SURVEY

Marlin randomizes the internal structure of executable code by shuffling the function blocks randomly and increasing the security of the system against code reuse attacks. In 2011 Tyler Bletsch et.al proposed a defense system, Control Flow Locking (CFL) against code reuse attacks. The control flow graph of the function cannot be deviated more than once, so it cannot be used for system call to attack the function. They perform a lock operation before each control flow transfer, with an unlock operation given at the valid destination of the function. The lock ensures the accuracy of the applications control flow. They insert a fragment of lock code in every indirect control flow transfer and the code avows the lock, by changing a certain lock value in memory. If the lock was already emphasized, a control flow violation will be detected and the program will be aborted or the execution passes through the control flow transfer to the destination. The Control Flow Locking reduces the code reuse attacks, reduces the performance meekly. They are set to reduce the code reuse attacks, which will reveal the programs control data [1].

In 2013 C.Zhang et.al suggested a new protection method CCFIR (Compact Control Flow Integrity and Randomization). They deal with the difficulties in CFI adoption. They collect the legal targets of indirect control transfer instructions and put in a specific springboard section in a random order and the gathered instructions will be pushed to the indirect control flow transfer. They validate the target with the springboard section and provide support for on-site-target-randomization. CCFIR

prevents control flow hijacking attacks such as ROP and return into 'libc' functions. There will be a chance to modify pointers that flow to external modules that are unprotected since CCFIR is applied only to parts of program [2].

In 2014 Y.Cheng et.al implemented ROPecker method, it defends all types of ROP attacks which do not need access to the source code. They use gadget chain detection algorithm to detect the chain in execution flow and sliding window mechanism triggers algorithm in proper time. The ROP attacks are of two phases: offline preprocessing phase and runtime detection phase. The instruction of the protected applications and shared libraries will be extracted and stored in database in the offline preprocessing phase. In runtime detection phase the events that trigger the detection logic will be executed using the sliding window and uncertain system calls will be processed. The application triggers page fault in sliding window phase and they will try to execute the code from the non-executable pages. The ROPecker module finds the relevant faults using the process ID and page fault error code. They verify the request and the pass it to the kernel and then invoke the ROP checking algorithm, thus guaranteeing there is no ROP gadget chain in the current stack of instructions [3]. In 2011 L.Davi et.al proposed a tool, ROPdefender which detects the conventional ROP attacks which are based on return instructions. The ROPdefender inspects the instruction type, during the execution of instruction by the processor. They identify the return address violations and prevent ROP attacks. Detection of all buffer overflow attacks over write return addresses of the instructions. They use instrumentation to scrutinize ROP attacks that are performed during runtime or compile time. The dynamic binary instrumentation is executed for avoiding access to side information [4].

In 2009 P.Chen et.al devised and employed deROP(Detecting Return Oriented Programming) for eradicating return oriented programming from instances of malware and malicious instructions. They enable malware analyzers, which in turn induces other malware analysis tools to scrutinize ROP based malware. The semantics of the original malware will be preserved in deROP, since it is fully automated. deROP requires execution of vulnerable application dynamically so that the gadgets that can be attacked by ROP can be identified. They emphasize dynamic analysis does not involve executing any malicious instructions in the original ROP exploit code. The limitation of deROP is that its output may differ from the traditional shell code [5]. In 2012 V.Pappas et.al proposed In-place code Randomization (IPR) technique which offers probabilistic protection against ROP attacks. They approach on narrow scope modifications in code segments of executable code by using an array of code transformation techniques. They apply the transformations statically and modify code for safely extracting from compiled binaries and they do not rely on symbolic debugging information. The modifications will not break the semantics of the code as the length of the instructions and basic blocks are being preserved. Also the randomization of stripped binaries without complete disassembly coverage will be enabled to avoid damage to the semantics. The purpose of this randomization process is to eradicate or probabilistically modify any number of gadgets which will be available in the address space of an exposed process. ROP code relies accurate execution of all chained gadgets, if there is alteration in a few may result in ineffective ROP code. The place code transformations can be done using (i) Atomic Instruction Substitution (ii) Instruction Reordering (iii) Register Reassignment In-place code randomization [6].

In 2013 L.V.Davi et.al explains the software diversity tool XIFER. XIFER exactly mitigates code reuse attacks assorting the structure of the application for each run. The binary rewriting will be done at the load time of the application. Binary rewriter is the main part XIFER, it disassembles binary application easily and performs code transformations and assembles new application instances with new memory layouts. The XIFER seek to ideal randomization tool in order to achieve instruction granularity randomization. They randomize all sections of executable information and the library diffuses fractions of executable segments to ensure they do not stay put in one block. By changing the code and data the leaked pointers cannot be used to calculate relative addresses of instruction. The randomization will take place dynamically thus it will not require an off line static analysis. They are not open to disclosure attacks as the diversification is applied again for each application run. XIFER provide memory overhead, as the possibility to write out ELF executable or shared library files will increase the file size [7].In 2012 R.Wartell et.al describes the Self Transforming Instruction Relocation (STIR); which is fully automatic and binary centric solution. They do not need any source code or symbolic information for target binary program. Every time STIR is launched the code reorders the basic blocks in each binary code section randomly, thus disallowing the attempts for predicting the location of the gadgets. They ensure that there will be no modification to the operating system or the compiler. A new binary is implemented and the basic block addresses are determined dynamically at load time. The system is fully transparent and it is enabled to self randomize legacy codes. The main components in STIR are: a conservative disassembler which transforms a target binary in to a randomized representation and a lookup generator. The address map of the randomizable representation will be encoded to the new binary. The lookup table generator and a load time reassembler will have the address map of the randomizable representation

encoded in it. The performance overhead will be reduced with the static code transformation approach [8].

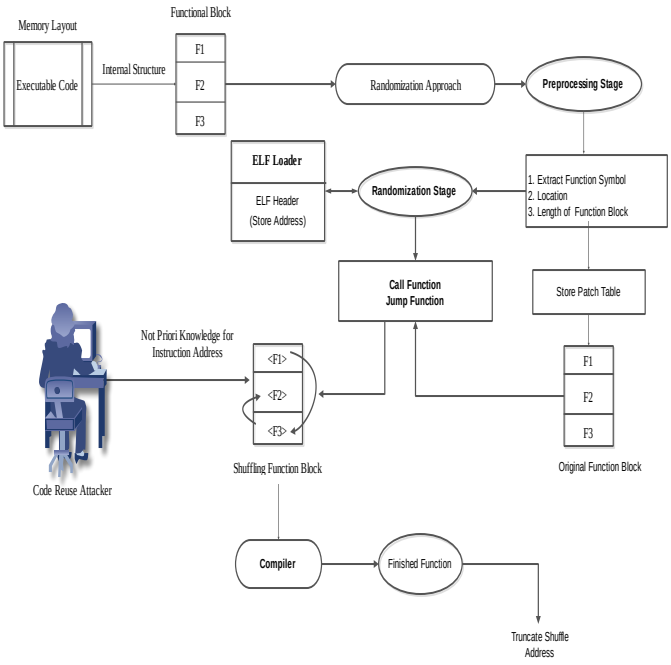
In 2012 J.Hiser et.al explains a new technique Instruction Location Randomization (ILR). This technique can be easily and efficiently applied to binary programs. Every instructions location in the program will be randomized in ILR, preventing the attacker to reuse program functionality. They operate on arbitrary executable programs and will not require compiler support without user interaction. Post deployment ILR can be automatically functional and eases frequent re-randomization. The preliminary prototype that works on 32-bit x86 Linux ELF libraries are described and it provides a high degree of entropy. They consign the individual instructions within a 31-bit address space randomly. They are not practical for the attacks that rely on prior knowledge of the location code or derandomization. The ILR is cost-effective and most realistic mitigation technique [9].

In 2010 T.Bletsch et.al proposed a new class of code reuse attack, Jump Oriented Programming (JOP). They eliminate the dependence on the stack and ret instructions. They build and chain the functional gadgets, with each of them executing certain primitive operations. The attack depends on the dispatcher gadget for dispatching and executing the functional gadgets. They use a dispatch table to hold the address and data of the gadget. The virtual program counter is maintained and the JOP program is executed by progressing it through the gadget. The entry of the functional gadget into the dispatch table is spotted with the help of program counter. All functional gadgets executed by the dispatcher must conclude by jumping back, for maintaining control of execution so that the next gadget can be initiated [10]. In 2009 R.Hund et.al proposed that, protecting the kernel of an operating system against attacks, specifically injection of malicious code. It is an important factor for implementing secure operating system. The design and implementation of the system automates the process of constructing instruction sequences which can be used by attacker for malicious computations. The kernel must be protected from the malevolent attacks. The basis of this mechanism is called as reference monitor, and it controls all accesses to the system resources and grants access to the verified systems. Kernel module signing provides the kernel integrity protection mechanism. Kernel code integrity can be achieved using this technique. Every kernel module must contain embedded and valid digital signature that can be checked against a trusted root certification authority (CA), if the kernel module signing is enabled. Loading of the code fails if this verification fails. The basic security guidelines that are to be followed by kernel code software developers will be established with the help of kernel module signing [11].

In 2009 L.Davi et.al devised a system to mitigate return oriented programming attacks. This system proposes a new runtime integrity monitoring technique which uses tracking instrumentation of program binaries that are based on taint analysis and dynamic tracing. Dynamic Integrity Measurement Architecture (DynIMA) is used to employ these techniques. This framework offers load time and runtime integrity for the program binaries and their source code location is not revealed even under the attacks of return oriented programming. The load time integrity measurement is combined with dynamic tracking techniques in DynIMA. The programs code will be loaded with the tracking code that presents integrity related runtime checks. The tracking code must contain new component that will rewrite the code of programs that is to be loaded for including special tracking code that monitors dynamic events of program and tracking data is maintained. The tracking code will be in the program binaries generically since we look to track common patterns of ROP attacks. So the source code of program need not be monitored by the DynIMA [12].

3.1 Proposed system and Objectives

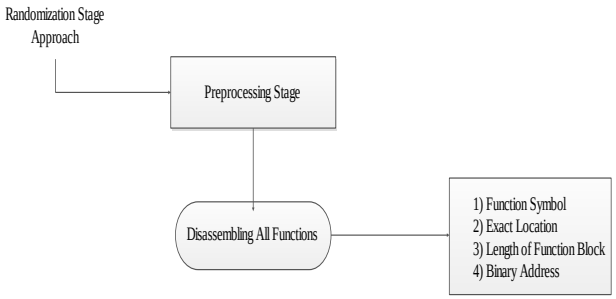
The code reuse attack makes use of the existing code of the system for malicious purpose. They make assumptions and hold the information about the memory layout of the executable code. The executable binary code will be shuffled with every execution of the binary in the randomization technique of Marlin. They shuffle the binary code at the function level and this coarse level granularity will not give any chance for brute force attack.



The target application will be randomized in Marlin before the control is conceded to the application for execution. There are two stages for randomization: preprocessing and randomization.

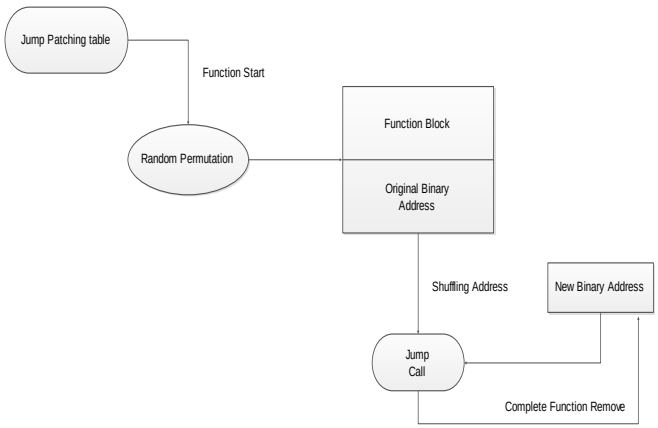
3.2 Preprocessing Stage

The binary address will be disassembled and the information about the function blocks is extracted in the preprocessing stage. The symbol, location, length and binary address of the function blocks are gathered for each function.



3.3 Randomization Stage

Randomization will be done in jump and call stages. The function blocks will be shuffled with respect to certain random permutation in the jump stage. A record of the original address of the functions and the new address where the function will exist in after the randomization of the binary will be maintained during the time of shuffling. The information will be stored in the jump patching table and it is discarded prior to the process where the binary is given control. In the call stage, the actual jump patching is executed and for every jump the jump patching table will be examined.



4. CONCLUSIONS

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. Thus it makes the brute force attack infeasible. The compiler will be confused for executing randomized instructions if their target binaries are obfuscated, so recursive traversal algorithm is implemented in sequencer to avoid confusion.

REFERENCES

[1] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in Proc. 27th Annu. Comput. Security Appl. Conf., New York, NY, USA, 2011, pp. 353–362.

[2] C.Zhang, T. Wei, Z. Chen, L. Duan, L.Szekeres, S. McCamant, D. Song, and W.Zou, "Practical control flow integrity and randomization for binary executables," in Proc. IEEE Symp. Security Privacy, 2013, pp. 559–573.

[3] Y.Cheng, Z. Zhou, M. Yu, X. Ding, and R. Deng, "ROPecker: A generic and practical approach for defending against ROP attacks," in Proc. 21st Annu. Netw. Distrib. Syst. Security Symp., 2014.

[4] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks," in Proc. 6th ACM Symp. Inf. Comput. Commun. Security, 2011, pp. 40–51.

[5] P.Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "DROP: Detecting return-oriented programming malicious code," in Proc. 5th Int. Conf. Inf. Syst. Security, 2009, pp. 163–177.

[6] V.Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in Proc. IEEE Symp Security Privacy, 2012, pp. 601–615.

[7] L.V.Davi, A.Dmitrienko, S.N€urnberger, and A.-R. Sadeghi, "Gadge me if you can: Secure and efficient ad-hoc instructionlevel randomization for x86 and arm," in Proc. 8th ACM SIGSAC Symp. Inf. Comput. Commun. Security, 2013, pp. 299–310.

[8] R.Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in Proc.ACMConf. Comput. Commun. Security, 2012, pp. 157–168.

[9] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd my gadgets go?" in Proc. IEEE Symp. Security Privacy, 2012, pp. 571–585.

[10] T. Bletsch, X. Jiang, and V. Freeh, "Jump-oriented programming: A new class of code-reuse attack," Dept.

Comput. Sci., North Carolina State Univ., Raleigh, NC, USA, Tech. Rep. TR-2010-8, 2010.
[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] L.Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks," in Proc. ACM Workshop Scalable Trusted Comput., 2009.