IJITKM 0973-4414 Volume-11, Issue-1 June-Dec 2017 pp. 9-14 **Impact Factor 2.8** available online at www.csjournalss.com

Preventing Buffer Overflow Using Buffer Linear Congruential Generator

Preeti Sharma, Monika Poriye, Vinod Kumar
M.Tech. Student, Assistant Professor
Deptt.of Computer Science and Applications,K.U. Kurukshetra (haryana)

Abstract: Buffer overflow is known to be a commonly found memory vulnerability that affects the software. In spite of continuing research, software vulnerability in software are still discovered and exploited. The protection techniques are suffered from many problems and one of them is high run-time overhead. C and C++ are generally more vulnerable to this attack because of unmanaged execution and out of boundary functions. To overcome the above- stated problem, a new technique is proposed that is data randomization using linear congruential random number generator.

Keywords: Buffer overflow, Memory errors, Randomization, Stack Randomization.

I. Introduction

Memory errors are continued to be the major problem in the field of security vulnerabilities such as buffer overflow, buffer underflow, double frees and dangling pointers etc. These errors are routinely exploited to gain the control over the execution of programs or to disclose information.
C and C++ are still vulnerable to above stated exploits due to some reasons such as:

- Writing after the end of array(i.e. called buffer overflow)
- Dereference of dangling pointer
- Use of some dangerous API functions that internally overflow a buffer (e.g. strcpy() , gets() etc.) and the use of printf() , scanf() in an unsafe way.

In this paper, buffer overflow problem is to be considered in which an attacker tries to store extra data than that buffer was intended to hold. This result in the overflow of adjacent memory regions, corrupt or overwrite the valid data stored in buffer. There are many variants of buffer overflow attack like stack overflows, heap corruption, format string attacks, and integer overflow and so on.
Although ASR and ISR are very much effective for preventing memory exploits in the past. But as the vulnerabilities are increased day by day, Buffer overflow problem is not captured by Address Space Randomization (ASR) .Some of the reasons behind unsuccessful ASR is as follows:
(i) Location of code and data segment can be randomized with PIE that causes performance overhead.
(ii) Kernel modification required.
(iii) Low entropy: Heap 13 bit, stack 24 bit.
On the other hand, Instruction Set Randomization (ISR) generally prevents the code injection by encoding with proper key. But ISR is also failed to solve the problem of buffer overflow problem because:
(i) Every instruction must be decoded at run time.
(ii) Most implementations use an emulator i.e. bochs or valgrind.
As both ASLR and ISR are vulnerable to attacks, Data randomization technique is used to randomize the data objects. The benefits of data randomization are:
(i) High entropy : 32 bit for integer data
(ii) Each data object may be protected using the different random value.
(iii) Effectiveness against non control data attacks not just pointer corruption.
(iv) Not vulnerable to information leakage attack.
(v) Can address intra-structure overflow.

II. Literature Survey

Address Space Layout Randomization

ASLR is a security technique that prevents memory exploitations. ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries. Address space randomization hinders some types of security attacks by making it more difficult for an attacker to predict target addresses. For example, attackers trying to execute return-to-libc attacks must locate the code to be executed, while other attackers trying to execute shell code injected on the stack have to find the stack first. In both cases, the system obscures related memory-addresses from the attackers. These values have to be guessed, and a mistaken guess is not usually recoverable due to the application crashing.
Address space layout randomization is based upon the low chance of an attacker guessing the locations of randomly placed areas. Security is increased by increasing the search space. Thus, address space randomization is more effective when more entropy is present in the random offsets. Entropy is increased by either raising the amount of virtual memory area space over which the randomization occurs or reducing the period over which the randomization occurs [1]. The period is typically implemented as small as possible, so most systems must increase VMA space randomization.

Instruction Set Randomization

Instruction Set Randomization (ISR) has been proposed as a promising defense against code injection attacks. It defuses all standard code injection attacks since the attacker does not know the instruction set of the target machine. In a code injection attack, an attacker exploits a software vulnerability (often buffer overflow vulnerability) to inject malicious code into a running program. Since the attacker is able to run arbitrary code on the victim's machine, this is a serious attack which grants the attacker all the privileges of the compromised process. [2]
By changing the instruction set, ISR defuses all code injection attacks. ISR does not prevent all control flow hijacking attacks, though; for example, the return-to-libc attack does not depend on knowing the instruction set. Much work has been done on the general problem of mitigating code injection attacks, and ISR is one of many proposed approaches.

StackGuard

StackGuard is a compiler extension that enhances the executable code produced by the compiler so that it detects and thwarts buffer-overflow attacks against the stack. The effect is transparent to the normal function of programs. The only way to notice that a program is StackGuard-enhanced is to cause it to execute C statements with undefined behavior: StackGuard-enhanced programs define the behavior of writing to the return address of a function while it is still active. StackGuard prevents changes to active return addresses by either detecting the change of the return address before the function returns, or by completely preventing the write to the return address. Detecting changes to the return address is a more efficient and portable technique, while preventing the change is more secure. [3]StackGuard supports both techniques, as well as adaptively switching from one mode to the other.

PointGuard

PointGuard is partially similar to data randomization but it only xors pointers and uses the same mask to xor all pointers. Therefore, it cannot prevent attacks that exploit memory errors to access non-pointer data and leaking any pointer value compromises the entire system. It may also fail to work on programs where pointers may be aliased with non-pointer data. [4]

Data Space Randomization

Data randomization involves four components: static analysis, compile-time instrumentation, load-time instrumentation, and run-time randomization. An example is taken to illustrate the results that is inspired by an real time attack on SSH Server [5]as shown in fig.1

```
1: void ProcessConnection(connection *c) {
2: cred_t user;
3: char message[1024];
4: int i = 0;
5:
6: auth_user(&user, c);
8: while (!end_of_message(c)) {
```

```
9: message[i] = get_next_char(c);
10: i++;
11: }
12: seteuid(user.user_id);
13: ExecuteRequest(message);
14 :}
```

Figure 1: Example code: simplified remote shell server with a buffer overflow vulnerability.

- **Static Analysis**

The Phoenix compiler framework is used to implement the static analysis that computes equivalence classes for data randomization. The analysis operates on Phoenix's medium level intermediate representation (MIR), which is still independent of the target processor [6]

- **Compile Time Instrumentation**

After computing the equivalence classes and their masks, the data randomization compiler generates code with instrumentation to encrypt and decrypt memory accesses. A Phoenix plug-in is implemented to insert the instrumentation. Since the static analysis works on MIR, the code is instrumented by transforming MIR. This avoids the complexity of mapping instruction operands between different code representations. Transforming a lower level intermediate representation would provide more control over the generated code, but the current version of the static analysis does not work on lower level intermediate representations. [7]

- **Load Time Instrumentation**

We generate new random masks when a program is loaded. To enable efficient re-assignment of masks to classes, the compiler emits a file with the byte offset, size of each immediate operand containing a mask, and the mask used. The loader uses this information to patch the loaded binary: it reads the old immediate value of a mask, looks up the corresponding new value, and overwrites the old value with the new one in the binary.

- **Run Time Instrumentation**

The runtime environment for data randomization provides an initialization function and wrappers for library functions and operating system calls. Our compiler inserts a call to the initialization function at the beginning of main. This function xors global variable and the arguments to main with the appropriate masks. Many attacks make use of libraries when exploiting vulnerabilities. For example, string manipulation functions are notorious for their use in exploits of buffer overflow vulnerabilities. [8] To increase data randomization's coverage, we provide wrappers for C library functions and operating system calls that receive or return pointers.

III. Proposed Work

Many techniques have been proposed to protect C and C++ programs from memory error exploits. Several tools analyze the source code to find the vulnerabilities but they are not sufficient because of imprecise results and raising false alarms.

Many memory safe dialects of C such as Clured and Cyclone that prevents the entire memory exploit but the limitation with these dialects is that they require the modification to the source code.

The general working steps of proposed work is as follows:

Step 1.) Take input from the user.

Step 2.) Store the input in message buffer.

Step 3.) Generate random numbers using linear congruential method.

Step 4.) Xor the random number with the input value.

Step 5.) Store the xored value in message buffer.

Step 6.) Apply instrumentation to provide the read and write integrity.

Here, Phoenix compiler framework is used to implement the static analysis that computes equivalence classes for data randomization. The analysis operates on Phoenix's medium level intermediate representation (MIR), which is still independent of the target processor [9].

The points-to sets are used to partition instruction operands into equivalence classes. Two operands are placed in the same equivalence class if they can refer to the same object at runtime according to the points-to analysis. This constraint ensures that instrumentation does not change program behavior in executions, it does not violate memory safety because all read and writes to an object are xored with the same mask. Under this constraint, number of equivalence classes must be maximized to increase the number of attacks that we can prevent. After that, random mask is generated by linear congruential method and then assign random value with each equivalence class [10].
At last, instrumentation is applied that provides with read and write integrity when the data is read from the buffer is in encrypted form and similarly data write onto the buffer is in decrypted form. Hence the analyzed results are shown in fig.1 as old without using random mask generator and fig 2 with using random mask generator

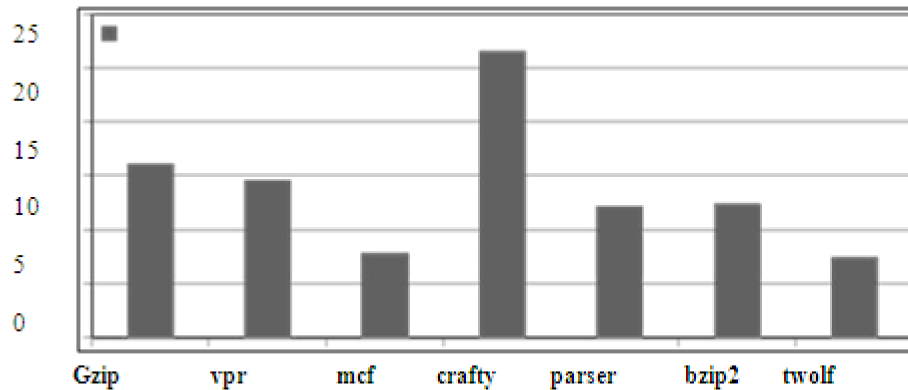

Fig.1 Data randomization without using linear congruential generator [11]

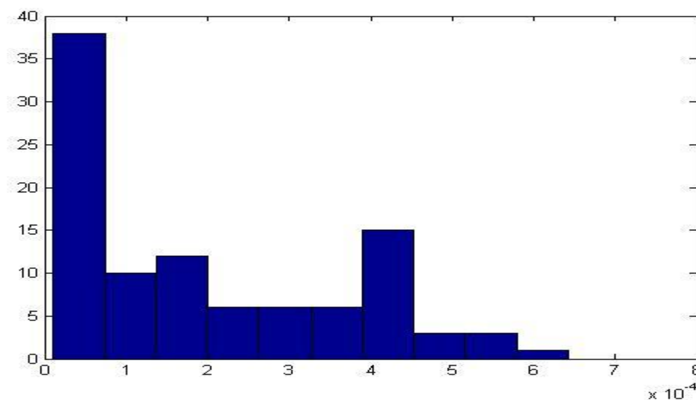

Fig.2 Data Randomization using linear Congruential generator

When data randomization is applied without using random mask generation method, the results are predictable and having high performance overhead. On the other hand, when the results are interpreted using linear congruential method then it reduces the probability of attack because of unpredictable values and less performance overhead.

I. Effectiveness

To evaluate the effectiveness of data randomization at preventing attacks, a benchmark is used with synthetic exploits and several exploits of real vulnerabilities in existing programs are used [12]. This section describes the programs and the vulnerabilities. It presents an analysis of the security afforded by data randomization.

Synthetic exploits

This run the control-data attacks that exploit buffer overflow vulnerabilities. The attacks are classified according to the technique they use to overwrite control-data, the location of the buffer they overflow, and the control-data they target. There are two techniques to overwrite control-data. The first overflows a buffer until the control-data is over-written. [13]The second overflows a buffer until a pointer is overwritten, and uses an assignment through the pointer

to overwrite the control-data. The attacks can overflow buffers located in the stack or in the data segment, and they can target four types of control-data: the return ad-dress on the stack, the old base pointer on the stack, function pointers and longjmp buffers in the stack or in the data segment. Table 6.1 shows that data randomization can prevent all the attacks in the benchmark

Real vulnerabilities

In final experiment, Ability of data randomization is tested to prevent attacks with a set of real vulnerabilities in real applications: SQL server, Ghttpd, Nullhttpd, and Stunnel [14].

SQL server is a relational database from Microsoft that was infected by the infamous Slammer worm. The vulnerability exploited by Slammer causes sprintf to overflow a stack buffer. Data randomization prevents the attack because the wrapper for sprintf randomizes the data that overwrites the current stack frame, including the return address. This causes the server to exit when freeing a local variable that was overwritten.
Ghttpd is an HTTP server with several vulnerabilities. The vulnerability that we chose is a stack buffer overflow when logging GET requests inside a call to vsprintf. Data randomization prevents the attack because the wrapper for vsprintf randomizes the value written by the attacker into the return address, causing the server to crash when the return address is used.
Nullhttpd [15]is another HTTP server. This server has a heap overflow vulnerability that can be exploited by sending HTTP POST requests with a negative content length field. These requests cause the server to allocate a heap buffer that is too small to hold the data in the request. While calling recv to read the POST data into the buffer, the server overwrites the heap management data structures maintained by the C library. [16]This vulnerability can be exploited to overwrite arbitrary words in memory. The attack works by corrupting the CGI-BIN configuration string. This string identifies a directory holding programs that may be executed while processing HTTP requests. Therefore, by corrupting it, the attacker can force NullHttpd to run arbitrary pro-grams. This is a non-control-data attack because the attacker does not subvert the intended control-flow in the server. Data randomization prevents the attack because the wrapper for recv randomizes the values written over the heap management data structures. This causes the server to crash when the values are used.

Stunnel [15]is a generic tunneling service that encrypts TCP connections using SSL. Format string vulnerability is studied in the code that establishes a tunnel for SMTP. An attacker can overflow a stack buffer by sending a message that is passed as a format string to the vsprintf function. Data randomization prevents the attack because the wrapper for vsprintf randomizes the value written by the attacker into the return address.

Table 6.1 Real attacks detected by data randomization. [17]

Application	Vulnerability	Exploit
NullHttpd	heap-based buffer overflow	overwrite cgi-bin configuration data
SQL Server	stack-based buffer overflow	overwrite return address
STunnel	format string	overwrite return address
Ghttpd	stack-based buffer overflow	overwrite return address

II. Conclusion

Data randomization using linear congruential generator is a new technique that provides probabilistic protection from memory error exploits. Data randomization uses static analysis to partition memory accesses into different classes according to the objects that they may access. Data randomization then assigns a distinct random mask to each class and, at runtime, xors the data read/written from/to memory with the corresponding mask. Therefore, memory accesses that violate the results of analysis, i.e., that access unintended objects, have unpredictable results. The results show that data randomization can block a broad range of attacks, while introducing an average runtime overhead up to 7% and an average space overhead below 1%.

References

[1] Hovav Shacham et al., "On the Effectiveness of Address Space Randomization," in 11th ACM Conference On Computer And Communications, 2004, pp. 231-236.
[2] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis, "Countering Code Injections With instruction set randomisation," in 10th ACM conference on Computer and communication security, 2003.
[3] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, and Peat Bakke, "StackGuard: Automatic Adaptive Detection And Prevention Of Buffer Overflow Attacks," in 7th USENIX Security Symposium San Antonio.
[4] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle, "Point-Guard: Protecting pointers from buffer overflow vulnerabilities," in 12th USENIX Security Symposium, 2003.
[5] SSH CRC-32 Compensation Attack Detector Vulnerability. securityfocus.com. [Online]. http://www.securityfocus.com/bid/2347
[6] Sandeep Bhatkar,
[7] T. Chiueh and F. Hsu, "A compile-time solution to buffer overflow attacks.," ICDCS.
[8] Etienne Kneuss, Philippe Suter, and Viktor Kuncak, "Runtime Instrumentation for Precise Flow-Sensitive Type Analysis," , 2010.
[9] Sandeep Bhatkar and R. Sekar, Data Space Randomization., 2010.
[10] http://wikipedia.org/linear-congruential-generator
[11] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Phillipe Martin, and Miguel Castro, "Data randomization.," Microsoft Research, USA, Technical Report MSR-TR-2008-120 2008.
[12] Oscar Hernandez, Fengguang Song, Barbara Chapman, and Jack Dongarra, "Performance Instrumentation and Compiler Optimizations for MPI/OpenMP Applications," University of Houston, Computer Science Department, USA, 2004.
[13] J. Wilander and M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention," in In Network and Distributed System Security Symposium (NDSS), pp. 149–162.
[14] D. Moore, V. Paxson, S. Savage, and C. Shannon, "Inside the Slammer," IEEE Security and Privacy, 2003.
[15] securityfocus.com. [Online]. http://www.securityfocus.com/bid/3748.
[16] Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code ," PLDI, 2001.
[17] Microsoft. Phoenix compiler framework. microsoft.com. [Online]. http://research.microsoft.com/
[18] SPEC. Spec Benchmarks. http://www.spec.org.
[19] Timur Iskhodzhanov, Reid Kleckner, and Evgeniy Stepanov, "Combining compile-time and run-time instrumentation for testing tools," In Programmnye produkty i sistemy, vol. 3, pp. 224-231, 2013.
[20] Vivek Iyer, Amit Kanitkar, and Partha Dasgupta, "Preventing overflow attacks by memory randomization," IEEE, 2010.