2010 IEEE 21st International Symposium on Software Reliability Engineering

Preventing overflow attacks by memory randomization

Vivek Iyer, Amit Kanitkar
Microsoft Corp.

Partha Dasgupta,
Raghunathan Srinivasan
Arizona State University
Tempe, USA
{partha, raghus}@asu.edu

Abstract—**Buffer overflow is known to be a common memory vulnerability affecting software. It is exploited to gain various kinds of privilege escalation. C and C++ are very commonly used to develop applications; due to the efficient "unmanaged" executions these languages are not safe. These attacks are highly successful as every executing copy of a shipped binary is the same. This work presents two approaches to randomizing the memory layout which does not require modifications at the developer end. Both techniques are implemented at the user-end machines and have no requirement for source code. The feasibility of the two techniques is shown by randomizing complex applications and demonstrating that the run-time penalty for the randomization schemes is very less.**

Keywords: Buffer overflow, stack randomization, heap randomization, software diversity

I. INTRODUCTION

The magnitude of the threat from malware especially on "consumer computing" platforms is well known and well understood. Attacks on a computing platform are possible due to the loopholes and bugs that exist in software. Buffer overflow attacks form a major genre of memory-errors related exploits in which a malicious user of a software program feeds data to a fixed length buffer that extends beyond the size of the buffer. An overflow can be used to overwrite the values in the stack; this can cause the program to return to an attacker specified location to execute attacker intended code. Buffer overflow has been popular for two decades [1]. It has been documented that buffer overflows constitute more than 50% of major security bugs [2]. C and C++ are very commonly used to develop applications; due to the efficient "unmanaged" executions these languages are not safe. A vast majority of vulnerabilities occur in programs developed with these languages [3]. Every memory buffer used in these languages is vulnerable to the overflow attack. The responsibility to perform run-time bounds checking is entirely placed on the developer. If proper bounds checking on a memory buffer are not in place, buffer overflow vulnerability is induced in the code. There are several variants of the buffer overflow attacks like stack overflows, heap corruption, format string attacks, integer overflow and so on [4]. When input data exceeds the amount of memory allocated to a memory buffer it overflows in the memory region adjacent to this buffer. This can potentially corrupt pointers in the region and cause run-time memory exceptions, program termination, and malicious code execution if cleverly crafted by an attacker.

In this paper two orthogonal approaches are presented; each of which provides a level of assurance against buffer overflow attacks. The approaches can be used independently or along with each other. This work focuses on providing "software diversity" for legacy software on end-user machines. Currently a malware designer can perform offline analysis of an application to discover vulnerabilities in it. These vulnerabilities can be exploited to launch various kinds of attacks on multiple systems. Every copy of an application that is shipped to consumers is exactly the same and contains the same weaknesses in the same binary locations. Software diversity breaks up the uniformity making each instance of the application different; consequently attacks that work on one instance do not work on another. Address Space Layout Randomization (ASLR) [5] is an example but this idea is taken to finer degrees of granularity on the stack frames and heap memory chunks in this paper.

The first approach focuses on stack-smashing class of buffer overflow attacks. A call stack is the dynamic data structure that keeps track of the active sub-routines of a program. Call stack is composed of stack frames corresponding to each active function call at any instant of program execution. A typical stack frame consists of: arguments passed to the function, return address to the caller, saved registers, and local variables of the function. It is easy to overwrite the return address on the stack frame by overflowing the buffer and storing data beyond its boundary. Carefully placed values can lead to the process executing code intended by the attacker. A post-compilation stack frame randomization technique is implemented in this work that minimizes the probability of stack smashing attacks and decreases the "return of investment" for attackers. The stack frame is randomized by re-writing specific binary instructions that access locations on the stack. The existing instructions are modified inside the binary without adding any new instructions or extra bytes of op-code.

The second approach presented in this work is heap randomization. Without proper bounds checking user data written into a heap memory block can overwrite the management data of next memory block. Under normal circumstances this would raise a memory exception and terminate the process, however, carefully placed data written into the heap management region of the next block can allow code to be written into a specific area in memory. This can allow an attacker to execute arbitrary code. Implemented in this work is a fine-grained heap randomization technique based on run-time modification of the heap memory allocations. For every block of memory requested a memory chunk with a size larger than requested is returned. The returned memory chunk contains random pads of memory appended before and after the actual memory to be used by the process. This is achieved

This material is based upon work supported in part by the National Science Foundation under Grant No. CNS - 1011931. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

1071-9458/10 $26.00 © 2010 IEEE
DOI 10.1109/ISSRE.2010.22

IEEE computer society

by static or run-time patching of heap memory management libraries in the operating system. This approach does not require changes to be made to compilers, linker, loaders or even existing application binaries.

Both methods have been implemented completely in software without the need for source code from software vendors. In the remainder of the paper, work related to these approaches and implementation details of the randomization approaches are presented.

II. RELATED WORK

A number of different strategies have been proposed to mitigate stack smashing attacks. Storing a duplicate copy of the return address for checking prior to return from a function has been proposed [6]. StackGuard [7] places a canary value between the local buffer(s) and the return address. It detects the occurrence of a buffer overflow based on whether or not the canary value is corrupted during the function execution. An advanced version of StackGuard known as ProPolice [8] is an extension developed for the GNU C Compiler. A static analysis of source code for identifying potential buffer overflow vulnerabilities has also been proposed [1, 9]. PointGuard [10] is another popular technique for protection against buffer overflows intending to corrupt pointers. PointGuard obfuscates pointer values by encrypting through an exclusive-or operation with a random value, the pointers are decrypted only when brought to CPU for dereferencing. Dynamic loadable libraries have also been implemented that do sanity checking on the return address of vulnerable library functions [11].

Randomization is a technique to inject diversity into computer software. The first known randomization of stack frame was proposed by placing a pad of random bytes between return address and local buffers [12]. Random pads make it difficult to predict the distance between buffers and the return address on the stack. An attacker has to launch custom attacks for every copy of the randomized binary. Address obfuscation has extended the above idea by randomizing the base address of memory regions, permuting the order of elements in a binary, and introducing random gaps within memory regions [13]. Most techniques used for stack randomization require compiler support. However the current software distribution model is such that only executable files are provided to the consumer. These files are already compiled and optimized; compiler techniques to randomization would not work on executable files. A wide deployment of compiler supported randomization would result in many consumers having the same 'randomness' in their binaries. The approach in this paper randomizes executable code by re-writing the instructions that allocate the stack frame in every routine in a binary file. This provides every consumer the opportunity to obtain a unique copy of the binary where the probability of the run time stack frame of two versions of the same binary being the same is low.

ASLR [5] modifies an operating system so that the base address of various segments of a program is randomly relocated during the program execution. Two such popular ASLR implementations named PaX and ExecShield are available for Linux as kernel patches. These approaches do not require the modification of individual binaries but necessitate that these binaries be compiled with a feature called Position Independent Executables (PIE). ASLR has been incorporated into Windows systems starting with Vista. It has been shown that PaX ASLR only marginally slows down the time taken to attack a system. Techniques have been demonstrated to bypass ASLR protection and to perform a de-randomization attack on known buffer overflow vulnerabilities [14, 15]. Two important differences between the stack frame randomization technique presented in this work and ASLR are that, firstly ASLR does not change the executables as it is a run-time randomization scheme, and secondly ASLR provides coarse randomization where the starting base address of the stack is randomized every time a binary is executed by the operating system, the relative addresses within the stack segment do not change. In this paper the stack frame randomization technique randomizes every stack frame by re-writing the instructions inside the binary. This provides fine grained randomization that requires no run time support. Importantly it also randomizes the relative addresses such that the distance between a local variable and the return address is not the same for different installations of the binary.

An approach to heap randomization [16] modifies the heap chunk structure to ensure the integrity of the allocated heap chunk. Extra bytes of data in the heap chunk structure are introduced which contains the header checksum seeded with a random value. The randomization seed is stored as a static variable. The checksum value is also used as a canary. Under this scheme allocation of a memory chunk to a process causes the computation of its header checksum, hence the checksum is placed as the canary. When this memory chunk is to be freed, re-computation of its header checksum occurs which is then compared with the existing checksum. If the two values do not match, an exception is raised and the process is terminated.

Service Pack 2 for Windows XP provides two major enhancements to the RtlHeap security not found in the predecessor versions of Windows. The first enhancement is the introduction of an 8-bit security cookie in the allocated chunk boundary tag to improve heap security, but as the security cookies is just 8-bits long it can hold only 256 unique values and can be brute-forced in applications which have a deterministic behavior [3, 17]. The second improvement provided by XP service pack 2 is the introduction of pointer sanity checking while performing the unlink operation. If the sanity checks are not satisfied, a heap corruption is indicated. However, even these techniques have been evaded in practice [17, 18].

Heap randomization for Windows has been implemented by hooking the *RtlHeapCreate*(), *RtlAllocateHeap*(), *RtlReAllocateHeap*(), and *RtlFreeHeap*() functions in *ntdll.dll* to provide 19 bits of randomness to base address of a newly created heap [19]. While this solution provides a high level of heap protection the authors themselves state that brute force attacks might succeed in evading their solution.

Address Space Layout Permutation (ASLP) [20] permutes, re-orders individual sub-routines contained within the code segments in addition to random relocation of code and data segments. It provides sub-page randomization for the stack and heap region by introducing random gap pages in between

used memory pages. ASLP changes the cross-references of the relocated routines for which it has to rely on the relocation information from the compiler.

III. STACK FRAME RANDOMIZATION

This section describes the stack frame randomization technique implemented in this work. The size of the run-time stack frame of every routine is randomized to ensure every version of a binary has a unique memory layout. The binaries are instrumented by analyzing the disassembly of the code segment in it. Additional bytes of code are not injected in the binary in this approach; instead existing bytes in the code segment are rewritten to achieve stack randomization. The randomization process is carried out after the application binary is installed at the end-user machine. During randomization only those instructions that are relevant to the run-time stack have to be rewritten. By shifting the vulnerable character buffer down by a random amount, the distance between the return address and the buffer becomes different for every copy of the binary. This makes it difficult to use the same attack string against different copies of the binary.

Randomization can be implemented at various levels. It can be done at the source code of the program. This would involve the addition of additional dummy variables in subroutines during compilation, making each copy of the binary different in terms of the stack frame. Another implementation may involve randomizing the code at assembly instruction level before using the assembler to build the binary. Randomization can also be achieved after the assembler has produced the object code or the executable binary. One of the basic assumptions made in this work is that access to source code would not be available. It is also assumed that the binaries themselves do not have any more information than available in a commercial grade fully optimized executable binary. Compiler support is not required for this work. A compiler patch makes it necessary to recompile source code to achieve randomization, whereas distribution binaries optimized for performance are randomized in this work. Similar techniques have been attempted in research for stack frame randomization [12, 13]. The difference between this work and existing research is the approach taken to randomize stack. Only the disassembly of a binary (executable) is utilized to randomize the stack frame. The output of the disassembler is parsed to separate the sub-routines within the code segment. This allows filtering of instructions that are related to the run time stack. The filtered machine-level instructions are located in the executable binary and instrumented to introduce a pad of random size into each function. This randomizes binaries without compromising its expected behavior or execution semantics. It is very important not to drastically alter the existing software distribution model. Eliminating the constraint of source code access provides the opportunity for randomizing legacy software. The randomization process is carried out at the user end. The distribution of randomization overhead to the users makes it critical that the randomization procedure is simple, low cost, and efficient. From the perspective of an attacker it can be assumed that access to the binary would be required to reverse engineer the randomization. This work is targeted for scenarios where attackers remotely target users using either a discovered or unknown/un-patched vulnerability in software, and do not already reside on the machine.

```
int foo(int dummy_arg)
{
 char mesg_string[1024];
 int local_variable;
 gets(mesg_string);
 local_variable = dummy_arg
}
```

Figure 1. Sample C routine

Consider the code shown in Figure 1. The routine *foo* takes in one integer argument. It has two local elements for which the compiler would need to allocate space on the stack. The compiler needs to allocate a buffer of 1024 bytes and 4 additional bytes for the integer variable. The C library function *gets* (known to be vulnerable to write beyond buffer bounds) is used to take input from the user to store in the character buffer. This makes the character array vulnerable to a buffer overflow attacks. Figure 2 shows the disassembly for the x86 architecture based binary of the function *foo*. The virtual address of each instruction is shown on the left, followed by the hex dump of the instruction op-code and operands. The rest of the disassembly shows the mnemonic instruction interpretation of the disassembler.

Instructions of the following type need to be rewritten in the binary if a random pad is added:

- Instructions that create space on the stack frame for local variables and buffers. The instruction at address 0x80483c7 in fig. 2 allocates space of 418 Hex bytes on the stack. If a pad of 32 bytes is injected into the frame, this instruction will have to allocate 438 Hex bytes of stack.
- Instructions that de-allocate space used by the locals of the function on the stack frame. These instructions are executed right before the function returns. In this case the stack de-allocation is done implicitly by the leave instruction that restores the stack pointer to the frame pointer; hence explicit modification of any instruction for correct de-allocation of the random pad is not required. In certain functions, a constant value is added to the stack pointer, to de-allocate space on the stack. Such functions require processing of de-allocation instructions.

```
080483c4 <foo>:
80483c4: 55                      push %ebp
80483c5: 89 e5                   mov %esp,%ebp
80483c7: 81 ec 18 04 00 00       sub $0x418,%esp
80483cd: 65 a1 14 00 00 00       mov %gs:0x14,%eax
80483d3: 89 45 fc                mov %eax,-0x4(%ebp)
80483d6: 31 c0                   xor %eax,%eax
80483d8: 8d 85 fc fb ff ff       lea -0x404(%ebp),%eax
80483de: 89 04 24                mov %eax,(%esp)
80483e1: e8 2a ff ff ff          call 8048310 <gets@plt>
80483e6: 8b 45 08                mov 0x8(%ebp),%eax
80483e9: 89 85 f8 fb ff ff       mov %eax,-0x408(%ebp)
80483ef: 8b 45 fc                mov -0x4(%ebp),%eax
80483f2: 65 33 05 14 00 00 00    xor %gs:0x14,%eax
80483f9: 74 05                   je 8048400 <foo+0x3c>
80483fb: e8 30 ff ff ff          call 8048330
                                 <__stack_chk_fail@plt>
8048400: c9                      leave
8048401: c3                      ret
```

Figure 2. Disassembly of sample C routine

- Instructions that access local variables and buffers on the stack frame. All local variables and buffers are accessed with the help of the frame pointer EBP. All stack locals are located below the frame buffer at lower addresses in the Intel x86 architecture. Due of the introduction of a random pad the local buffers shift further down from the frame pointer. The instruction at 0x80483e9 in figure 2 accesses a location 1032 bytes below EBP, this instruction will need modification.

A naive coarse-grained randomization technique would be to modify every relevant instruction with the same random pad constant. This would alter the size of the stack frame of every function or sub-routine by the same random amount. A coarse-grained technique makes it simpler for an attacker to launch a brute force attack by trial and error technique. In order to make it difficult to launch such brute force attacks, a fine-grained randomization of stack frames is essential. Uniform randomization also poses another problem where a constraint that prevents randomization of one function will make it impossible to randomize other unrelated portions of the code segment.

Constraints: Integer constants are represented in the form of two's compliment in a binary file. An N-bit two's compliment can represent integers in -2^{N-1} to $+ 2^{N-1}$. The amount of padding that a routine can be provided is dependent on the instruction that can support the least padding. Consider the instruction mov %edx,-0x80(%ebp). In this instruction, the negative constant offset from the frame pointer is -0x80. With a single byte signed integer -0x80 is the most negative number that can be represented. This instruction would have to be modified when a random pad is provided to the function. The local variable would move further away from the EBP making it necessary to make the operand more negative. An extra byte is required to represent the new integer even if a pad of one byte is provided to this function. Hence the function in which this type of instruction is present cannot be provided even one byte of pad. However such constraints apply only to a specific sub-routine as every routine is processed separately.

Certain functions require usage of several general purpose registers for internal computation. Such functions may preserve the values of the registers that get clobbered as they are restored before the function returns to the caller. Such functions have a specialized function prologue and epilogue to accomplish this. Figure 3 shows the function prologue and epilogue of one such function. Such specialized function epilogues use negative offsets along with the EBP to calculate the effective address. The instruction at 80578cb deceivingly looks like an access to a local buffer; such instructions in function epilogues should never be randomized. Any modification to this instruction would potentially corrupt the stack frame pointer causing either a segmentation fault or some other run-time memory exception leading to a system crash.

Certain instructions cause general-protection exceptions when the memory operand they operate on is not aligned as per the architecture specification. One such instruction is FXSAVE which saves the floating point context to a 512-byte memory location specified in the 16-byte aligned operand. The addition of a random pad can violate this constraint, causing the instruction to result in a general-protection exception or an alignment check exception. Any stack frame randomization technique should be careful while dealing with such sensitive instructions. A conservative approach is to not randomize sub-routines that have these instructions.

```
8057520: 55                    push %ebp
8057521: 89 e5                 mov %esp,%ebp
8057523: 57                    push %edi
8057524: 56                    push %esi
8057525: 53                    push %ebx
8057526: 81 ec bc 03 00 00     sub $0x3bc,%esp
..............
..............
80578cb: 8d 65 f4              lea-0xc(%ebp),%esp
80578ce: 5b                    pop %ebx
80578cf: 5e                    pop %esi
80578d0: 5f                    pop %edi
80578d1: 5d                    pop %ebp
80578d2: c3                    ret
```

Figure 3. Disassembly of stack restoration of EBP offset

Adding a constant to the stack pointer ESP is one of the methods to restore its value at the end of function execution. Such instructions should be rewritten during randomization as instructions that create space on the stack are randomized. There is another scenario in which a constant is added to the stack pointer to restore the ESP after a subroutine calls other functions during the course of its execution. Arguments to sub-routines are typically pushed onto the stack before the call instruction is executed. When the called function returns it is the responsibility of the caller to clean the arguments off the stack frame and reset the stack pointer. In order to accomplish this, the caller adds a constant to the ESP that represents the number of bytes pushed onto the stack as arguments to the function that just returned. Such instructions are capable of misleading a randomizer into acting on it as an instruction that restores the ESP at the end of function execution. This may lead to corruption of the stack pointer, resulting in segmentation faults or run-time memory exceptions.

The frame pointer (EBP) is used for access to local variables on the stack. The random padding injected at the beginning of the frame makes it necessary to modify instructions that use EBP to access local variables. The frame pointer is also used to access the arguments passed to a function. The difference between these two kinds of instructions is based on whether a positive or a negative offset is used in the instruction. In the x86 architecture, the local variables are located at lower addresses below the frame pointer so negative offsets are used with EBP to access them. The arguments are located above the frame pointer at a higher address and positive offsets are used to access them. Only the local variables shift during randomization; hence the random pad does not affect the arguments on the stack. EBP references using positive offsets that access arguments to a function should not be randomized.

Implementation: The prototype randomizer was developed in C and compiled using gcc, the *objdump* utility was used to perform disassembly. Figure 4 shows the flow of the implemented randomizer. The binary file is fed to the disassembler; its output is parsed for identification of instruction operands that need to be modified in the binary. The parser separates out and analyzes each sub-routine in order

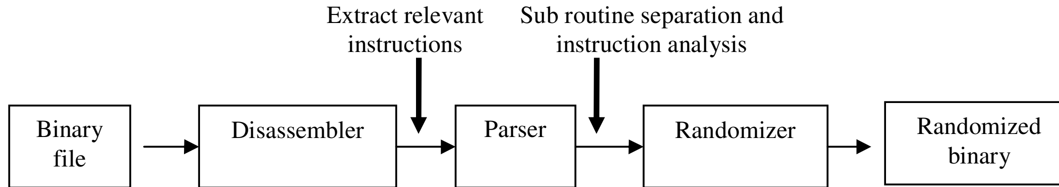

Figure 4. Workflow of randomizer

to accomplish fine-grained randomization such that every function is padded separately with a random pad conforming to the constraints of that specific function. These instructions are then rewritten directly into the binary.

The prototype works as a 2-pass randomizer. In the first pass each sub-routine is analyzed to determine the maximum padding that can be provided to that routine. The randomizer walks through every instruction within a sub-routine and updates a data structure that is associated with that routine. This data structure has the following information: the virtual address of the routine, the maximum pad that can be provided to the routine, and the random pad assigned to the routine. The randomizer looks for instructions that are sensitive to alignment of memory operands and takes a conservative approach of not randomizing the subroutine containing them. Due to the way the Intel architecture binary accesses stack memory, the amount of randomization that can be performed is limited. Every instruction in a function is processed to check the maximum possible random pad allowed for that instruction. The least of these values becomes the available max pad ($\mathfrak{p}_{Max}$)for the function. The actual random pad is ($\mathfrak{p}_{Rand}$) selected by generating a random number between 0 and $\mathfrak{p}_{Max}$. $\mathfrak{p}_{Rand}$ is then clipped to the nearest factor of 32 to resolve many alignment requirements of several instructions. It is necessary to place an upper-limit on $\mathfrak{p}_{Max}$ provided to each subroutine. Providing extremely large pads to all the subroutines increases the chances of a stack overflow which can cause the process to crash. An upper-limit of 640 bytes on the pad for every subroutine is maintained to prevent this scenario.

In the second pass the randomizer goes through the instructions in the disassembly and locates them in the executable binary file. While tracing every instruction the randomizer keeps track of the subroutine in which the instruction is present. With the help of the data structure built for every subroutine during the first pass the randomizer statically rewrites and instruments the corresponding instruction in the binary executable. It can be argued that this solution is vulnerable to brute force techniques where an attacker could attempt to overflow the stack by providing one specific return address at every stack location.

Using this technique an attacker can successfully jump into a library call or a different function instead of returning to the calling routine. However, passing arguments to such an overflow attack can be considered very difficult. This is due to the fact that many library calls read arguments off the stack and these arguments have to be correctly placed on the stack. As a result of this even if an attacker executes a brute force attack where the execution returns to a library function, it would be impossible to pass arguments. In addition if this technique is used in addition to using canaries the runtime system can quickly determine that an attack occurred. The prototype randomizer was tested on a variety of binaries. All the binaries used in testing were release quality optimized utilities that are part of Linux distributions. Binaries of different kinds ranging from moderate to complex were used in testing. Copies of the following applications were randomized in this work: Open Office, pidgin, pico, ls, gcc, netstat, ifconfig, route, xcalc, tail, date, nslookup, sum, head, wc, md5sum, pwd, users, cat, cksum, hostid, logname, echo, size, firefox, viewres, xwininfo, oclock, ipcs, pdfinfo, pdftotext, eject, lsmod, clear, vlc, and gij. A small list of binaries that were successfully randomized is shown in Table 1. The binaries were thoroughly tested for segmentation faults and other runtime memory errors that may

TABLE I. PERFORMANCE TEST RESULTS FOR STACK RANDOMIZATION

Binary	Time taken to randomize (sec)	% of subroutines randomized	Original stack (bytes)	Instrumented stack (bytes)	Overhead (bytes)
Open Office	20.220	88.33	21256	23170	1,914
pidgin	54.718	91.48	16093	16407	314
gcc	1.836	91.77	1829	2009	180
route	0.505	93.75	2183	2614	431
xcalc	0.109	100	11,372	11592	220
echo	0.137	100	1018	1079	61
firefox	1.200	100	27890	28216	326
vlc	0.057	100	6366	6688	322
gij	0.062	100	2467	2633	166
lsmod	0.397	100	4414	4473	59

```
Virtual      Disassembly of                  Disassembly of
Address      Original Routine                Instrumented routine

8049d80:     push %ebp                       push %ebp
8049d81:     mov %esp,%ebp                   mov %esp,%ebp
8049d83:     sub $0x10,%esp                  sub $0x30,%esp
8049d86:     mov %edi,-0x4(%ebp)             mov %edi,-0x24(%ebp)
8049d89:     mov 0x8(%ebp),%eax              mov 0x8(%ebp),%eax
8049d8c:     mov 0xc(%ebp),%edi              mov 0xc(%ebp),%edi
8049d8f:     mov %ebx,-0xc(%ebp)             mov %ebx,-0x2c(%ebp)
8049d92:     mov %esi,-0x8(%ebp)             mov %esi,-0x28(%ebp)
8049d95:     mov 0x38(%eax),%esi             mov 0x38(%eax),%esi
8049d98:     mov 0x38(%edi),%ecx             mov 0x38(%edi),%ecx
8049d9b:     mov 0x34(%eax),%ebx             mov 0x34(%eax),%ebx
8049d9e:     mov 0x34(%edi),%edx             mov 0x34(%edi),%edx
8049da1:     cmp %ecx,%esi                   cmp %ecx,%esi
8049da3:     jl 8049dc6 <exit@plt+0x356>     jl 8049dc6 <exit@plt+0x356>
8049da5:     jle 8049dc2 <exit@plt+0x352>    jle 8049dc2 <exit@plt+0x352>
8049da7:     cmp %ecx,%esi                   cmp %ecx,%esi
8049da9:     movl $0x1,-0x10(%ebp)           movl $0x1,-0x30(%ebp)
8049db0:     jle 8049dd0 <exit@plt+0x360>    jle 8049dd0 <exit@plt+0x360>
8049db2:     mov -0x10(%ebp),%eax            mov -0x30(%ebp),%eax
8049db5:     mov -0xc(%ebp),%ebx             mov -0x2c(%ebp),%ebx
8049db8:     mov -0x8(%ebp),%esi             mov -0x28(%ebp),%esi
8049dbb:     mov -0x4(%ebp),%edi             mov -0x24(%ebp),%edi
8049dbe:     mov %ebp,%esp                   mov %ebp,%esp
8049dc0:     pop %ebp                        pop %ebp
8049dc1:     ret                             ret
```

Figure 5. Comparison of Original and Instrumented Disassembly

occur due to corruption of the runtime stack randomization. None of the binaries exhibited any deviation from their expected behavior and no run-time exceptions were generated. On an average the randomizer modified the run-time stack of more than 75% of the sub-routines in every application. This was due to the conservative approach taken by not randomizing functions containing certain instructions like FXSAVE. Other constraints mentioned also contribute to this number being less than 100%.

Figure 5 shows a sample randomization instance where the original routine contained an instruction to generate 16 bytes of stack. After passing the routine through this prototype randomizer it can be seen that the stack allocated is 48 bytes. As can be seen in the figure, no net instruction is added or removed; only the instructions that access the local variables on the stack are modified to point to the correct location on the randomized stack frame which in this instance has a shift of 32 bytes. The experiments show that the randomization cost itself is reasonable and low. Only the size of the run-time stack was manipulated, hence this approach is not expected to have any runtime penalty.

IV. HEAP MEMORY RANDOMIZATION PER ALLOCATION

Heap memory randomization was achieved by over-allocating the requested chunk and then placing the returned chunk within the over-allocated chunk. As a first step the library functions that play a vital role in heap memory management (functions that perform the free, allocate, and resize operations) were isolated. These functions were wrapped with randomization code. Access to the source code of an application was not used for this solution as the solution patches the underlying heap memory management mechanism. A dual random padding strategy was used for every memory allocation. This was done by appending a random pad below and above the pointer to the heap memory chunk returned by the allocation routine. Figure 6 gives an overview of the randomization technique. Randomization ensures that each heap allocation request originating from a program receives a different buffer during different instances of execution. The internal layout of the process heap looks different on every run. This approach mitigates heap overflow attacks in deployed software. The randomization implemented in this work does not require changes to be made to the system loader, linker, and compiler. The randomization process does not modify

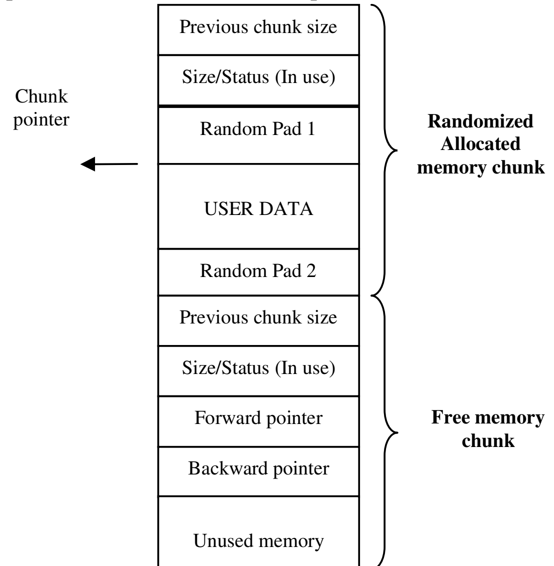

Figure 6. Allocated heap memory chunk with dual random padding

application binaries and does not impose any noticeable execution overhead on a process. Randomizing the starting location of every allocated buffer to obfuscate the boundary pointers is unique to the Linux solution presented.

Linux approach: The random pads placed on the heap memory are non-uniform and generated on-the-fly during runtime. The dual random padding strategy is implemented by appending a random pad below and above the pointer to the heap memory chunk returned by the allocation algorithm. This is achieved by identifying and patching the heap management functions of the GNU C library. The functions identified were: malloc(), free(), realloc() and memalign(). Other functions identified were calloc(), valloc() and pvalloc(), however these functions are based on malloc() and were not modified.

Two random integers i and j (both multiples of 8 to respect the internal memory alignment rules) are generated during an allocation request. The upper limit of the random numbers generated can be selected heuristically. These two random integers are added to the size parameter contained in the original request resulting in an allocation call for i + j + original request. A successful malloc() operation returns the pointer to a newly allocated memory chunk. The value of the pointer returned to the calling function (user application) is shifted by i bytes. This integer value i is stored just above the returned pointer so that memory management functions like free() and realloc() can determine the random pad value to calculate the actual starting address of the memory chunk and the boundary tag information stored above it.

A call to free() made by the requestor function, results in execution entering the free() public wrapper. The actual starting address of the allocated chunk is calculated in the wrapper and passed to the internal free() routine. If these values are not determined, it would lead to errors in the system. The value of random numbers set by malloc() is extracted by reading the location directly above the chunk address passed as argument to free. Using the integer values stored above the passed chunk address the original size of the block and the starting address of the block is determined. Similar calculations are made for realloc() and memalign() library calls.

TABLE II. RUN-TIME PERFORMANCE TESTS ON LINUX UBUNTU SYSTEM

Test	Execution time without the library patch (sec)	Execution time with the library patch (sec)
Dhrystone 2 with register variables	10	10.1
Whetstone Double Precision	10.2	10
Execl Throughput	29.3	29.8
File Read (buffer size = 1024 Max. number of blocks = 2000)	30	30
File Write (buffer size = 1024 Max. number of blocks = 2000)	30	30
File Copy (buffer size = 1024 Max. number of blocks = 2000)	30	30
File Read (buffer size = 256 Max. number of blocks = 500)	30	30.1
File Write (buffer size = 256 Max. number of blocks = 500)	30	30
File Copy (buffer size = 256 Max. number of blocks = 500)	30	30
File Read (buffer size = 4096 Max. number of blocks = 8000)	30	30
File Write (buffer size = 4096 Max. number of blocks = 8000)	30	30
File Copy (buffer size = 4096 Max. number of blocks = 8000)	30	30
Pipe-base Context Switching	10	10
Process Creation	30	30
System Call Overhead	10	10
Shell Script (8 concurrent)	60	60.2

TABLE III. RUN-TIME PERFORMANCE TESTS ON WINDOWS XP SYSTEM

Test	Execution time without the library patch (sec)	Execution time with the library patch (sec)
CPU – Integer Math (Mops/s)	54	69.5
CPU – Floating Point Math (Mops/s)	220.8	218.9
CPU – Prime Number Calculation (1000 primes/s)	245.3	241.9
CPU – Multimedia Instructions (Millimatrices/s)	1.36	1.32
CPU – Compression (Kbytes/s)	744.9	732.5
CPU – Encryption (Mbytes/s)	2.95	2.96
Physics (Mbytes/s)	40.7	38.9
String Sort (1000 strings/s)	591.2	592
Memory – Allocate Small Block (Mbytes/s)	933.7	1434.5
Memory – Read Cached (Mbytes/s)	1340.4	1334.2
Memory – Read Uncached (Mbytes/s)	1272	1271.2
Memory – Write (Mbytes/s)	812.4	814.1
Large RAM	120.1	118.5

Windows approach: The solution for Windows RtlHeap security also involves generating non-uniform random pads on-the-fly (during runtime) and adding them to a heap memory chunk to be allocated. The end results provided by this solution appear similar to the Linux solution; however Windows poses technical limitations to the implementation of the solution. The source code for Windows or its RTL (Run-time Library) is not available. Hence the internal workings of the required heap memory management RTL functions cannot be studied and it is not possible to edit them to recompile the library as done on a Linux system. This necessitates a different patching technique. The *Detours* [21] research package provided by Microsoft was utilized to implement heap memory randomization on Windows. *Detours* is a library which allows intercepting arbitrary Win32 binary functions on x86 machines. The *Detours* library achieves API hooking on the target function by overwriting few bytes in the beginning of the function's in-core binary image. First the *Detours* library injects the user-supplied library containing the 'hook' or the 'Detour' function into the target process' memory. Once this is done it copies over the first few bytes of the target function (to be overwritten later) into a newly allocated region known as a *trampoline* function. As its last instruction this *trampoline* contains a jump to the remainder of the target function. The first few bytes of the target function are overwritten with a jump instruction to the user provided 'Detour' function. This finishes the insertion of the 'Detour' function and the target Win32 function is *hooked*. All future calls made to the target function land in the 'Detour' function instead of the original function.

Using this run-time library patching or hooking technique the memory management functions are overridden like the Linux implementation. The kernel APIs in kernel32.dll (*HeapAlloc()*, *HeapFree()* and *HeapReAlloc()*) were hooked to achieve this. To implement location randomization, the ability to write into the user area of the returned heap chunk is required. In Windows attempts to save the values of the random integer i above the shifted pointer as implemented on Linux caused segmentation faults. To achieve randomization a separate heap request A2 is made in addition to the original request A1 where A2 provides the random padding of i to the allocated chunk request. To ensure that A2 is released during a free operation the address of A2 is stored in the last but one word of A1. The last word of A1 is placed with a "cookie" that indicates that the heap is randomized. *HeapFree() detour* function checks for the four byte cookie during every free operation request and only if the cookie is detected it frees the chunk A2. A double free request is prevented by the implemented *HeapFree() detour* function as it erases the *cookie* during the first free operation. The *HeapReAlloac() detour* function does not have any responsibilities as opposed to the corresponding one in Linux. In the absence of pointer location randomization all the randomization changes related to reallocation are implicitly handled by *HeapAlloc()* and *HeapFree()* detours.

Performance: Several release-quality applications were tested with and without the implemented library patch on Linux Ubuntu 8.04 and Windows XP operating systems. Applications such as web browsers, office suites, games, and graphics processing utilities were executed without any exceptions. These applications did not exhibit any unexpected behavior and no run-time exceptions were detected during the test runs. The Linux solution was evaluated on a benchmarking tool known as Unixbench (version 4.1); the results of these tests can be seen in table 2. On the Windows system, the solution was evaluated on a benchmarking tool known as Performance Test (evaluation version 7.0); the observations of the tests are reported in table 3. From the results of the benchmarking tools it is apparent that the heap randomization solutions do not induce any noticeable run-time overhead on the applications executed with the modified libraries.

V. CONCLUSION

Techniques in randomization of software have been researched heavily in the recent years. Randomization reduces

the return of investment for an attacker and makes it difficult to construct attacks that can be replicated. Buffer overflow attacks have been a security problem for nearly two decades. The stack frame randomization approach in this paper mitigates the prevalence of stack smashing attacks. The disassembly of executable binaries was utilized to separate, analyze, and rewrite instructions relevant to the run-time stack of a process. Fine grained randomization was performed for a number of binaries where each routine was randomized independently of other routines in the binary. Tests conducted using the randomized binaries showed that the process was a low cost approach that lends itself to the existing software distribution model without making large scale system wide modifications like changes to the compiler/linker/loader. Although this technique makes it difficult to launch a buffer overflow attack aimed at hijacking the control flow it does not protect against attacks that intend to corrupt data adjacent to vulnerable binaries. A better randomization technique can be developed based on this work that sprinkles pads of random sizes across a stack frame and within the data variables or locals of a function.

This paper also presented a fine-grained technique to randomize the heap section of a program. The library patching technique used in this solution introduces random pads above and below every chunk of memory allocated off the heap. The values for these random pads are generated at run-time due to which each instantiation of a program allocates different random pads to the same heap memory buffer. In Linux the GNU C library was statically patched to introduce random pads during heap allocation. Under Windows a patch library is injected into system which contains *detours* or patch functions for the target memory management functions. Hooking into these *detours* allows randomization of allocated heap buffers. Due to the technological limitations such as absence of source code, pointer location randomization is not a viable technique for Windows, however developing a workaround technique for the can be pursued as future work.

REFERENCES

[1] D. Wagner , J.S. Foster, E.A. Brewer, and A. Aiken, “A first step towards automated detection of buffer overrun vulnerabilities,” Networks and Distributed Security Symposium, pp. 3 – 17, 2000.

[2] J. Viega and G. McGraw, “Building Secure Software,” Addison-Wesley: 2002.

[3] R. Seacord, “Secure Coding in C and C++,” Addison-Wesley: 2005.

[4] J. Foster, V. Osipov, N. Bhalla, and N. Heinen, “Buffer Overflow Attacks: Detect, Exploit, Prevent,” Syngress Publishing: 2005.

[5] Web link, ASLR: “Address space layout randomization,” retrieved on April 25 2010, http://pax.grsecurity.net/docs/aslr.txt

[6] T. Chiueh and F. Hsu. Rad: "A compile-time solution to buffer overflow attacks," 21st International Conference on Distributed Computing Systems, pp. 409–417, April 2001.

[7] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks," SSYM’98: Proceedings of the 7th conference on USENIX Security Symposium, pp. 63–78, 1998.

[8] H. Etoh. "Propolice: GCC extension for protection against stack-smashing attacks," Available online at: http://www.trl.ibm.com/projects/security/ssp/.

[9] D. Larochelle and D. Evans. "Statically detecting likely buffer overflow vulnerabilities," SSYM’01: Proceedings of the 10th conference on USENIX Security Symposium, pp. 177–190, August 2001.

[10] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. "Pointguard: protecting pointers from buffer overflow vulnerabilities," SSYM’03: Proceedings of the 12th conference on USENIX Security Symposium, pp. 91–104, 2003.

[11] A. Baratloo, N. Singh, and T. Tsai. "Transparent run-time defense against stack smashing attacks," ATEC ’00: Proceedings of the annual conference on USENIX Annual Technical Conference, pp. 21–21, 2000.

[12] S. Forrest, A. Somayaji, and D.H. Ackley, “Building diverse computer systems,” HOTOS ’97: Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI), IEEE Computer Society, pages 67–72, May 1997.

[13] S. Bhatkar, D. C. DuVarney, and R. Sekar, “Address obfuscation: an efficient approach to combat a broad range of memory error exploits,” 12th USENIX Security Symposium, vol 120, August 2003.

[14] T. Durden. "Bypassing PaX ASLR Protection," Phrack Inc. Available from URL http://www.phrack.org/issues.html?issue=59&id=9#article.

[15] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. "On the effectiveness of address-space randomization," CCS '04: Proceedings of the 11th ACM conference on Computer and communications security, pp. 298-307, 2004.

[16] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur. "Run-time Detection of Heap-based Overflows," Proceedings of the 17th Large Installation Systems Administration (LISA XVII), 2003.

[17] M. Conover and O. Horovitz. "Windows Heap Exploitation (Win2KSP0 through WINXPSP2)," Available online: www.cybertech.net/~sh0ksh0k/heap/XPSP2%20Heap%20Exploitation.ppt.

[18] A. Anisimov. "Defeating Microsoft Windows XP SP2 Heap Protection and DEP Bypass," Available online: http://www.aelphaeis.milw0rm.com/papers/121.

[19] L. Li, J.E. Just, and R. Sekar. "Address-Space Randomization for Windows Systems," Proceedings of the 22nd Annual Computer Security Applications Conference 2006 (ACSAC'06).

[20] C. Kil, J. Jun, C. Bookholt, J. Xu, P. Ning. "Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software," Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC’06).

[21] G. Hunt and D. Brubacher. "Detours: Binary Interception of Win32 Functions," In 3rd USENIX Windows NT Symposium, USENIX, July 1999.