RAD: A Compile-Time Solution to Buffer Overflow Attacks

Tzi-cker Chiueh   Fu-Hau Hsu
Computer Science Department
State University of New York at Stony Brook
{chiueh, fhsu}@cs.sunysb.edu

Abstract

*Buffer overflow attack can inflict upon almost arbitrary programs and is one of the most common vulnerabilities that can seriously compromise the security of a network-attached computer system. This paper presents a compiler-based solution to the notorious buffer overflow attack problem. Using this solution, users can prevent attackers from compromising their systems by changing the return address to execute injected code, which is the most common method used in buffer overflow attacks. Return Address Defender (**RAD**) is a simple compiler patch that automatically creates a safe area to store a copy of return addresses and automatically adds protection code into applications that it compiles to defend programs against buffer overflow attacks. Using it to protect a program does not need to modify the source code of the protected programs. Moreover, RAD does not change the layout of stack frames, so binary code it generated is compatible with existing libraries and other object files. Empirical performance measurements on a fully operational RAD prototype show that programs protected by RAD only experience a factor of between 1.01 to 1.31 slow-down. In this paper we present the principle of buffer overflow attacks, a taxonomy of defense methods, the implementation details of RAD, and the performance analysis of the RAD prototype.*

1: Introduction

This paper presents a solution to the notorious buffer overflow attack problem. Using this solution, users can prevent attackers from compromising their systems by changing the return address to execute injected code, which is the most common method used in BO attacks. Anecdotal evidence shows that BO attacks have already been used to attack programs since the 1960s [18]. The most famous BO attack is the Internet Worm written by Robert T. Morris in 1988 [17]. Buffer overflow attacks can inflict upon almost any kind of programs and is one of the most common vulnerabilities that can seriously compromise the security of a network-attached computer system. Usually the result of such an attack is that the attacker gains the root privilege on the attacked host.

Although the buffer overflow problem has been known for a long time, for the following reasons, it continues to present a serious security threat. First, programmers do not have the discipline to check array bounds in their programs and most compilers do not do this also thus programs with this vulnerability are generated continuously. It is not easy to ask all programmers to check array bounds in their programs. For example, as of the writing of this paper, July 23rd 2000, the title of one of the latest vulnerabilities reported by CERT [5] is " CA-2000-06 Multiple Buffer Overflows in Kerberos Authenticated Services." Secondly, not all applications with this vulnerability have been found and for those that have been found, it is not easy to replace all of them. For the above reasons, having a tool to seal this security breach automatically is very important.

Return Address Defender (**RAD**) is a compiler extension that automatically inserts protection code into application programs that it compiled so that applications compiled by it will no longer be hijacked by return address attackers.

Section 2 describes the principle of buffer overflow attacks and a taxonomy of defense methods. Section 3 describes the design and implementation details of RAD. Section 4 presents the effectiveness of RAD and its performance overheads. Section 5 reviews related works in this field. Section 6 is the conclusion.

2: Buffer Overflow Attacks

2.1: Principle of Buffer Overflow Attacks

If programs don't check the size of the user input for a buffer array and the size of the input data is larger than the size of the buffer array, then areas adjacent to the array will be overwritten by the extra data. The lack of such ``bound checks'' creates the breeding ground for buffer overflow attacks [3, 4, 20, 22].

Program variables with similar persistence properties are assigned into the same memory area. Within each area, variables' locations are adjacent to each other. Any writing past the bound of a data structure will overwrite adjacent data structures and change their values. If the overwritten area contains a function's return address, then when the function returns, this new value will be used as the address of the next instruction after the return. So if a hacker could inject his/her code into memory and change a return address to point to the injected code, then he/she can have the inserted code executed with the attacked program's privilege. Attackers can execute injected code by applying the same method to function pointer variables [23] as well.

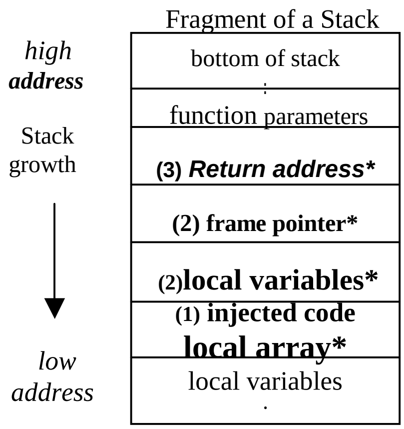

Figure 1: buffer overflow attack steps:
(1) Feed the buffer array with the injected code through any I/O statement in the attacked program. (2) Continue to feed the attacked program with injected string. (3) Overwrite the original return address with the new address pointing to the injected code. The regions marked with `*' are data structures on the stack that get overwritten by overflow attacks.

C compilers allocate space for local variables and the return address of a function in the same stack frame. Within each frame these objects' locations are adjacent to each other, as shown in Figure 1. Most C compilers do not perform array bounds checking. As a result, C programs' arrays become the favorite targets of buffer overflow attacks. In order to launch such an attack, all an attacker needs to do is to (a) compose a string containing his/her code and a return address pointing to the code, and (b) insert the string into the correct place in some stack frame of the attacked program through an I/O statement. Then when the function whose local buffer array is overwritten returns, the injected code is executed. Because the inserted code is executed with the attacked program's privilege, set-root UID programs and programs with root privilege, e.g. daemons, are attackers' favorite targets.

2.2: Defense Methods

Injecting malicious code and addresses into a victim program (step A), changing its control flow at run time (step B), and executing the injected code (step C) are the three essential steps to successfully launch a buffer overflow attack. A successful attack must have all of these 3 steps. Several different solutions have been proposed or implemented to solve the buffer overflow problem through preventing one or more of these 3 steps. According to the strategy they use, we can classify these defense methods into the following three categories.

The first type of defense methods defeats buffer overflow attacks by prohibiting the injection of malicious code. Richard Jones et al. [16] and OpenBSD [13] use this strategy to protect programs. In Jones's approach, they developed an extension to GCC to automatically perform array bounds and pointer checking. In OpenBSD, they manually inspect and modify the kernel source code to perform array bounds and pointer checking.

The second type of defense methods allows foreign code to be injected and even modifications to return addresses, but prohibits unauthorized changes of control flow. So attackers can inject their code into memory and can change return addresses, but control flow cannot be transferred to the injected code. RAD and OGI's StackGuard [1] are both based on this strategy. RAD uses RAR and StackGuard uses canary words to prevent injected addresses from being used as return addresses of function calls.

The third type of defense methods allows steps A and B to take place, but disables step C. So code and addresses can be injected into memory and control flow can be transferred to the injected code, but the injected code cannot be executed completely. Solar Designer's non-executable stack [14] and Sekar's [24] and Lee's [25] intrusion detection methods use this strategy to protect network applications from buffer overflow attacks. In Solar Designer's case, they make the stack non-executable; so even though control flow can be transferred to the injected code, the code cannot be executed. In Sekar's method, they manually build normal/abnormal behavior patterns in terms of system call sequences and their arguments for each program to be protected. By comparing the run-time behavior of the protected program with its known legitimate patterns, they can detect and prevent the attacks. Lee's method is based on a similar principle, but uses a data mining approach to build up the patterns dynamically. Both intrusion detection methods allow injected code to be executed, but attempt to detect abnormal behavior or known intrusion patterns to stop malicious code.

3: Return Address Defender

RAD is a patch to **gcc-2.95.2** that automatically adds protection code into the function prologues and epilogues of the programs compiled by it. So the source code does not need to be modified. By overflowing a return address with a pointer to the injected code, attackers can have the code executed with the attacked program's privilege. Return address defender (**RAD**) prevents this by making a copy of the function return address in a particular area of the data segment called *Return Address Repository* (**RAR**). By setting neighboring regions around RAR as read-only, we can defend RAR against any attempt to modify it through overflowing. Given that RAR's integrity is guaranteed, each time when a return address of a stack frame is used to jump back to the caller function, this address is checked with the copy in RAR. A return address will be treated as un-tampered and thus safe to use only if RAR also contains the same address.

There are two versions of RAD, MineZone RAD and Read-Only RAD, which protect the return addresses stored in RAR in two different ways. Both methods are portable. MineZone RAD is more efficient while Read-Only RAD is more secure.

3.1: MineZone RAD

In MineZone RAD, we create a C file, **/hacker/global.c**, and modify gcc-2.95.2, so the file is automatically linked with programs compiled by RAD. This C file contains all the function definitions and variable declarations used in the new function prologues and epilogues. In **global.c** we declare a global integer array and divide it into 3 parts as shown in figure 2.

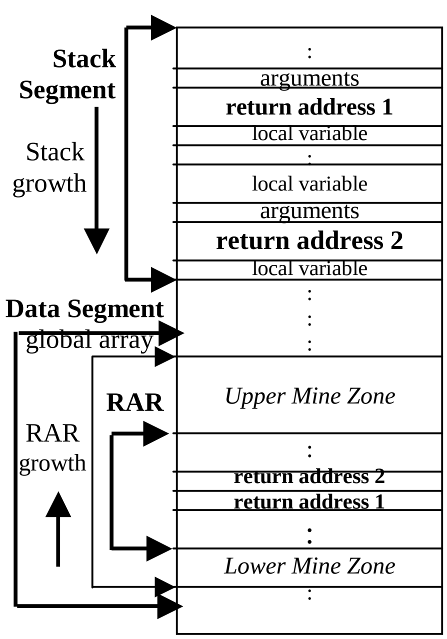

Figure 2 Structure of RAR and Mine Zones

The middle part of the global integer array is RAR, which keeps a redundant copy of the return address of each function call. The first and third parts, call *mine zones*, are set as read-only areas by **mprotect()** system call so that any writing attempts to these areas will cause a trap. All protection functionalities are implemented as instructions added to the new function prologues and epilogues without changing the stack frame layout of each function. Therefore programs compiled by RAD are compatible with existing libraries and other object files. In the new function prologue, the first instruction executed is "pushing a copy of the current return address into RAR." In the function epilogue, before returning to the caller, the callee compares the current return address with the top return address in RAR. If they are the same, RAD will pop it from RAR and go back to the caller. Otherwise this means someone is attacking the return address, and a real-time message and an email are sent to the system administrator and the attacked program is terminated. Figures 3 and 4 list the protection instructions' pseudo code.

```
Set areas around RAR as read-only
   /*  This  statement  is  executed  when  the
program starts and is executed only once. */
If RAR is full
 {  Send warning message to user;
    Terminate the program;
 }
Else
     Push current return address into RAR;
```

Figure 3: Function Prologue Code

MineZone RAD is a simple and efficient mechanism to protect return addresses, and it can survive attacks through buffer overflow to overwrite RAR or stack frame return addresses, which is the most common form of BO attacks. But if the attacked program satisfies several special conditions simultaneously, a Direct Return Address Modification Attack could still compromise the program even though it is protected by MineZone RAD. In the next subsection we present a variant of RAD that can resist DRAMA.

```
If (top return address in RAR== return address
in current stack frame)
{  Pop the top return address in RAR;
   Go back to caller;
}
Else
{  Send a real-time message and an email to the
   system administrator;
   Terminate the program; }
```

Figure 4: Function Epilogue Code

3.2: Read-Only RAD

If a program satisfies all the below statements and conditions simultaneously, attackers could launch a direct return address modification attack:

(1) A statement of the form *A=B; where A is a pointer variable and B is a variable.
(2) A loop statement that copies user input into a buffer array without checking its bounds, and the array is adjacent to variable A.
(3) A loop statement that copies user input into a buffer array without checking its bounds, and the array is adjacent to variable B.
(4) Before *A=B is executed, statements in (2) and (3) must have been executed. After statements in condition (2) and (3) are executed, there are no other statements that change A's and B's values.

Under the above conditions, attackers could launch an attack in the following way: Before *A=B is executed, use statements in conditions (2) and (3) in the attacked program to change A and B's values. Then after statement *A=B is executed, the address pointed to by the new value of A will get the new value of B. In fact, using this attack pattern, attackers can change the content of any memory location, including return addresses. However, so far we are not aware of any published exploit code that is based on this attack pattern.

Read-Only RAD is similar to Mine-Zone RAD. But instead of setting up mine zones, Read-Only RAD sets the RAR itself as read-only to protect itself. As in MineZone RAD, instructions preventing buffer overflow attacks are added to the function prologues and epilogues. Figures 5 and 6 list their pseudo code. In Read-Only RAD, RAR is set as read-only most of the time. The only time that it becomes writable is in the function prologues when the current return address is pushed into RAR. Since there cannot be any external input statements in the function prologue instructions, so DRAMA doesn't have any chance to change RAR. Of course, because RAR is set as read-only to update it in function prologues requires adding two extra system calls to each function call, causing a serious performance penalty.

```
Set RAR as writable;
If RAR is full
{  Send warning message to user;
   Terminate the program;  }
Else
    Push current return address into RAR;
Set RAR as read-only;
```

Figure 5 Function Prologue Code

```
If (top return address in RAR== return address
in current stack frame)
{  Pop the top return address in RAR;
   Go back to caller; }
Else
{  Send a real-time message and an email to the
   system administrator;
   Terminate the program; }
```

Figure 6 Function Epilogue Code

3.3: Inconsistency of Address Storage

So far RAD is based on the following assumptions:

- When a function is called, its stack frame is pushed into the stack and is not popped from the stack until it finishes and returns.
- When a function call returns, only its stack frame is popped.

These assumptions are not always true for C programs. System calls **setjmp()** and **longjmp()** [7] allow users to bypass several functions in the call path to the current function to directly jump back to the function executing setjmp(). Users use setjmp() to set a return location and use longjmp() in a different function to go back to the return location set by the setjmp() statement. If between the executions of setjmp() and longjmp() there are several nested function calls, then when longjmp() is executed, the execution goes back to the setjmp() statement directly. Consequently not only the current stack frame but also all stack frames between these two functions' frames are popped from the stack. So the top return address of RAR and the return address in the current stack frame do not match in this case. According to 3.1 and 3.2, if the mismatch occurs, RAD will treat this as a potential attack and terminate the program.

Because executing longjmp() will pop more than one stack frame, we can address this problem by simulating the above action by popping the return ad-dresses in RAR accordingly. When detecting that the top return address of RAR is different from the return address in the current stack frame, instead of terminating the program, RAD pops RAR and repeats the comparison. If no match can be found when RAR is empty, then it means someone is launching an attack, otherwise the matched return address is safe to use.

However, this scheme leads to another problem, as illustrated in Figures 7. For each stack frame, we only list its return address. a, b, c, d, e, f, and g are return addresses of functions A, B, C, D, E, F and G. In function A there is a setjmp() statement. In function F there is a longjmp() statement. Figure 7-(1) shows a particular calling sequence, and its associated stack and RAR layouts. In this calling sequence, function G calls function D, which in turn calls function A. Function A executes setjmp() and then calls function B which in turn calls function A. But this time function A does not execute setjmp() and then calls function E. Function E calls function F. In function F, the longjmp() statement is executed. After longjmp() is executed, the stack frames of F, E, A, and B are popped. The new stack layout is shown in Figure 7-(2). When function A returns, the stack layout is shown in Figure 7-(3). But using the new method, RAD only pops the return addresses of F, E and A, as shown in Figure 7-(3). Obviously the two layouts are differently. Will this inconsistency cause any problem? The answer is No! For this calling sequence, when function D returns to function G, the stack layout becomes the one in Figure 7-(4). Then RAD pops return addresses b, a, and d. So now they become the same again.

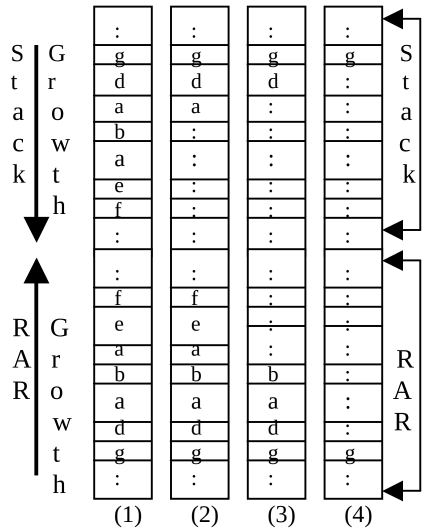

Figure 7 Layout inconsistency between RAR and stack due to the execution of longjmp()

RAR only holds legal return addresses and RAD only uses RAR to check whether a return address is legal. RAD never changes or sets the return addresses used in programs. RAD either terminates the execution of a program or lets the program continue. So the layouts of return addresses in RAR and in the stack do not need to be the same all the time. Besides, RAD guarantees that no illegal entry is added into RAR,(RAD uses MineZone RAD and Read-Only RAD to achieve this goal.), and when function call pushes a legal entry into RAR, the entry is still in RAR when the function whose prologue pushed the return address into RAR is about to return. So RAD works correctly. A formal proof of correctness of RAD is given in [26].

4: Evaluation of RAD

4.1: Implementation Issues

The code implementing RAD consists of two parts, a patch to gcc and declarations in /hacker/global.c. There are about 85 lines of C and GNU 80x86 in-lined assembly code in the patch part that inserts protection code into function prologues and epilogues and links global.c's object file, global.o, with other object files to generate the final executable file. There are about 90 lines of C and GNU in-lined assembly code in global.c that contains variables' and functions' definitions used in the new function prologues and epilogues.

The run time address space of a program compiled by RAD contains two copies of return addresses, one in the stack and one in RAR. Any attempt to change return addresses in the stack will be detected by RAD and result in the termination of the program and the delivery of a warning message to root. RAR is protected by system call mprotect(). mprotect()'s default reaction to memory access violation is a core dump. But this behavior could be changed. By making a patch to the kernel, RAD can react to memory access violation the same way as in the stack case. RAD uses a new system call to tell the kernel the exact portion of the address space that mprotect () uses to protect RAR. So when a memory access violation occurs, the kernel can use this information to determine whether it should make a core dump (due to a violation occurring in other mprotected-area) or terminate the program and send a read-time message to root (due to a violation occurring in the area used to protect RAR). In our case the patch to the kernel uses about 70 lines of C and GNU in-lined assembly code.

When the patched kernel detects an attempt to modify RAR, it uses system call exec() to execute the program /hacker/hacker, which sends a real time message to the root and terminates the program. System call exec() requires that its parameters be stored in the user address space, but now execution is within the kernel. The patched kernel solves this problem by putting the parameters into the stack of the user address space before invoking exec(), so exec()'s parameters will be in the right place when it is executed and kernel can execute /hacker/hacker correctly.

4.2: Effectiveness of RAD's Defense

Aleph One's exploit code [3] is the template of many exploit codes used to launch a buffer overflow attack against a variety of applications. Many exploit codes only add small modifications to it. So our first step in testing the effectiveness of RAD is to test whether RAD can defend Aleph One's code against the attack of his own exploit code. The test showed that RAD indeed stops the attack effectively. Originally, our next step is to find more exploit codes and then test them against their target victims with or without the protection of RAD. But for the following reasons, we chose a different way to evaluate the effectiveness of RAD. Currently those who attack return addresses only know that stack return addresses are their targets. They don't know that RAD keeps another copy of each function return address in RAR. So even if we could show that RAD can successfully detect existing buffer overflow attack codes, we still can not be sure that RAD can protect return addresses in all cases, because none of existing attacks know the existence of RAR, let alone attack it. Therefore, we chose to classify basic attack patterns, and evaluate RAD's defense efficacy based on how well it can handle these basic attack patterns.

Attack pattern 1 is the most common form of buffer overflow attacks. Aleph One's code uses this pattern. A program containing a loop statement that reads input from outside without performing array bounds checking is a genetic victim. To launch a pattern 1 attack to change a return address, attackers must start from a nearby location of the address and repeat writing toward the return address. So when the return address is overwritten, the area between the start point and return address is also modified. In MineZone RAD, both sides of RAR have a read-only mine zone to protect it. Any one trying to use pattern 1 to change a return address in RAR will touch the mine zone first, which will result in the termination of the attacked program. So MineZone RAD can handle pattern 1 attacks successfully.

If a program satisfies all the conditions listed in 3.2, which correspond to attack pattern 2, then even with MineZone RAD, attackers can still successfully hijack the program under attack, MineZone RAD can only prevent overflowed writing of RAR from the upper or lower directions, but cannot prevent direct random writes into RAR. In Read-Only RAD, except in function prologues, RAR is read-only and in function prologues there is no any I/O and loop copy statement, so attackers do not have any chance to change RAR. So this RAD protects return addresses from both attack patterns

4.3: Performance Evaluation

This section describes tests that we use to evaluate the performance overhead of RAD. OGI's work [1] provides a good way to find the upper bound of a function's additional performance cost of their scheme, so we adopt their methods to evaluate the performance overhead of RAD.

4.3.1: Micro-Benchmark Results

We wrote three C programs to evaluate three kinds of performance costs that RAD incurs. We use as the penalty metric the ratio of a function's additional performance cost associated with RAD to the function's original performance overhead. That is, *Penalty = A function's additional performance cost associated with RAD ÷ the function's original performance overhead* .

Each program has only two functions. Except the main function, there is another function that increases the value of a variable by one. We call the function 100,000,000 times and use **gettimeofday()** to measure the execution times required with or without the protection of RAD. All tests are made on a 133MHz Pentium processor with 32 Mbytes of main memory. In program 1, we call a function, void inc(),100,000,000 times to increment a global variable 100,000,000 times. The function has no arguments and no return value. In program 2, we call a function, void inc(int *), 100,000,000 times to increment a local variable of main() 100,000,000 times. The function has no return value, but has one argument that is the address of the local variable. In program 3, the function used is int inc(int) which increments one to one of its local variables. We call the function 100,000,000 times to measure RAD's performance penalty. The function has a return value and an argument. Table 1 and Table 2 show the micro-benchmark results of MineZone RAD and Read-Only RAD. All measured numbers are in terms of µs.

Because the overhead of a function call is fixed and the overhead of the additional instructions added by RAD is also fixed, the more computation a function performs, the less RAD's penalty is in term of relative percentages. In MineZone RAD, the additional over-head comes from the cost to store and manage return addresses in RAR. There are about 20 lines of assembly instructions in the function prologue and epilogue to perform the protection operation. In Read-Only RAD, not only the above protection operation needs be performed, but also two mprotect() system calls must be executed to protect RAR, which introduces a serious performance penalty. This is the reason why the performance of Read-Only RAD is much higher compared to MineZone RAR.

4.3.2: Macro-Benchmark Results

In subsection 4.3.1, we measure performance penalties on functions containing only one statement. They provide an upper bound on RAD's relative performance penalty. The real performance penalty of executing a RAD-protected program should be the sum of the penalties of all functions in the program. In this subsection we measure the performance penalties of two significant Unix applications, **ctags** and **gcc**. ctags is a UNIX command that generates an index file for several languages, including C. The version we used is ctags-3.2. Totally there are 9,488 lines of source code. The input program that gcc and RAD-protected gcc compile is a dictionary proxy server with 4500 lines of source code. For each program, we measured the execution times of the original program and the RAD-protected version. From these measurements, we calculated the performance penalty.

Function Prototype	Original run-time	MineZone RAD run-time	Penalty
Void inc()	12,814,378	30,897,126	1.41
Void inc(int *)	17,334,197	35,418,721	1.043
Int inc(int)	18,089,581	36,924,768	1.041

Table 1 Micro-benchmark results of MineZone RAD

Function Prototype	Original run-time	Read-Only RAD run-time	Penalty
Void inc()	12,814,378	2,696,119,743	209.40
Void inc(int *)	17,334,197	2,628,705,475	150.65
Int inc(int)	18,089,581	2,651,345,171	145.57

Table 2 Micro-benchmark results of Read-Only RAD

Program size	Program tested	User time	System time	Real time
11991 lines	Original ctags	0.57	0.05	0.62
	MineZone RAD-protected ctags	0.58	0.05	0.63
	Read-Only RAD-protected ctags	8.16	19.17	27.32

Table 3 Macro-benchmark results of ctags

Program size	Program tested	User time	System time	Real time
4500 lines	Original gcc	3.53	0.19	3.72
	Mine Zone RAD-protected gcc	4.67	0.2	4.87
	Read-Only RAD-protected gcc	20.46	50.43	70.89

Table 4 Macro-benchmark results of gcc

The measurement results are shown in Tables 3 and 4, and are all in terms of seconds. As the additional cost of a RAD-protected program comes from the additional instructions added to function prologues and epilogues. So the more computation a program's functions perform, the less performance penalty its RAD-enhanced version experiences. gcc's functions have more computation than ctags's so the penalty of Read-Only RAD-protected gcc, *18x*, is smaller than the penalty of Read-Only RAD-protected ctags, *43x*. But the number of functions executed also influences the *relative* performance penalty. RAD introduces a fixed extra performance cost for each function protected by it, so the more functions executed, the more additional cost is added. gcc has more functions executed than ctags does, so the penalty of MineZone RAD-protected gcc,*0.3x*, is larger than the penalty of MineZone RAD-protected ctags, *0.02x*.

5: Related Work

Crispin Cowan and Calton Pu et. al. [1] has developed a gcc patch StackGuard that protects return addresses from being modified to point to the injected code without the need to modify the source code. There are 3 variants of StackGuard, Canary version, MemGuard Register, and MemGuard VM.

The Canary version of StackGuard puts a canary word before every return address in the stack. By checking the integrity of the canary before a function returns, this version can defeat most of pattern 1 attacks with little performance penalty. But if attackers can correctly guess the canary value, this method will not work. Besides under certain conditions attackers could overwrite the pre-computed canary values with their own values to turn off the protection [19]. In addition, due to alignment requirement, it is possible to overwrite a return address but skip over the canary word. The Canary version can't survive pattern 2 attacks. Even though MineZone RAD also can't survive these attacks either, it is more difficult to hijack the program protected by MineZone RAD, because attackers must change both the return addresses in stack and in RAR. The biggest drawback of the Canary version is that it changes the layout of stack frames, because an additional canary is inserted into every stack frame. This change will result in unexpected behavior in some programs. According to OGI [21], ``the major compatibility limitation of StackGuard is that it changes the format of an activation record on the stack. Programs that are introspective with respect to the format of data on the stack will fail.''

Both MemGuard versions try to make the memory areas holding return addresses as read-only after they have been saved into those areas. MemGuard Register uses the 4 Pentium debug registers to monitor any write action into the return addresses in the top-most 4 stack frames. MemGuard VM achieves this by setting the stack as read-only. By installing a trap handler, they catch writes to protected pages, and emulate the writes to non-protected words on protected pages. These two versions are more secure than the canary version. But not every processor architecture supports debug registers. As MemGuard Register has used all debug registers, so programs compiled by it have no free debug register to use. Finally if the total size of the top-most 4 stack frames is less than one page and there are other stack frames in the topmost page that MemGuard Register does not set as read-only to reduce the performance penalty, then return addresses in these stack frames are without protection and vulnerable to return address attacks. The MemGuard VM version exhibits a very serious performance penalty due to frequent memory references to a read-only area.

Because system damage can only be inflicted via system calls, a running process's system call patterns are good targets to monitor to detect intrusions. R. Sekar et al. [24] defined the behavior patterns of process in terms of sequences of system calls and their arguments. Before a program is executed, these patterns are manually written into a specification with a special language, called Regular Expression for Events (REE). So correctly using ERR to accurately describe the program behavior is essential for this method to work. Wenke Lee et al [25] used a data mining approach on audit data to identify legitimate pattern of resource usage. So before the patterns are established a large amount of data must be collected for the intrusion detection system to learn.

Both strategies' effectiveness relies on the existence of correct system call patterns. So whenever a program is to be executed, not only the executable file but also the corresponding system call pattern must be available. Moreover, any changes to the protected applications may entail a corresponding change on the associated patterns.

Alexandre Snarskii has written a patch to libc to make FreeBSD un-exploitable with standard stack overflow attacks [15]. Functions containing statements that get inputs for buffer arrays from outside without making bounds checking are vulnerable to buffer overflow attacks. If a library function has this vulnerability, all programs using this function are vulnerable to buffer overflow attacks also. Strcpy(), strcat(), and sprintf() are such library functions. Snarskii made a patch to these dangerous library functions to check the stack integrity before they return to solve this problem. Because the patch is only for C library functions, vulnerabilities in user-defined functions still exist and continue to threaten the security of computer systems.

In an extension to the GNU C compiler, Richard Jones and Paul Kelly have developed a new method to enforce array bounds and pointers checking in the C language [16]. They make the check without changing the representation of pointers, so code with array bound checking is compatible with the unchecked version. In their approach, every pointer value is valid for only one memory region that could be a single variable, an array, a single structure, an array of structure, or a single unit of memory allocated by malloc(). For every pointer expression, they define a unique base pointer. Then by checking whether the memory region referenced by the result of the pointer expression is the same region as the one referenced by the base pointer, they enforce bounds checking. This method solves the buffer overflow attack problem, including those attacks targeting at function pointers. But the performance penalties are substantial. For a matrix multiplication, there is a 30x slowdown. Moreover, in their implementation, under certain conditions array bounds checking is disabled.

Injecting code into the stack and changing the return address to point to it is the most common form of BO attacks, so making the stack non-executable can effectively defeat the BO attack. Solar Designer [14], an alias, has written a patch for Linux kernel to make the stack non-executable. This patch has a very small performance cost. Besides, since it doesn't need to recompile the source code, users don't need to find the source code to use this new kernel. But this method only works for traditional and standard stack attacks. Exploit code injected into data segment still can hijack the attacked program.

Linux signal handler returns need an executable stack. Nested function calls and trampoline functions also need an executable stack to work properly. Functional languages, e.g. LISP, also need an executable stack. In order to solve the above problems, the patch needs to temporarily set the stack as executable when the above events occur. But this also creates a window of vulnerability for attackers to launch a buffer overflow attack.

6: Conclusion

This paper describes the design, implementation, and evaluation of a compiler solution called RAD to the buffer overflow attack problem. RAD is simple and effective, and does not suffer from the stack frame compatibility problem associated with previous approaches. In addition, RAD can correctly handle network application programs that use setjmp() and longjmp() calls. We have also presented a taxonomy of memory overwrite attack patterns, and show that the less secure version of RAD, MineZone RAD, can survive attack pattern 1, while the more secure version, Read-Only RAD, can survive all attack patterns at the expense of much higher performance penalty. Binary code generated by RAD is compatible with existing libraries and other object files. When RAD is used to protect a program, there is no need to modify the source code. Finally RAD will send a real-time message and an email to the system administrator when it detects an attack. From the prototype implementation and performance measurements we believe RAD provides an effective way to protect computer systems against buffer overflow attacks.

Acknowledgment

This research is supported by NSF MIP-9710622, NSF IRI-9711635, NSF EIA-9818342, NSF ANI-9814934, NSF ACI-9907485, USENIX student research grants, as well as a contract from Siemens.

Reference

[1] Crispin Cowan, Calton Pu, et.al. , ‘‘StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks,’’ Proceedings of the 7th USENIX Security Conference, San Antonio, Texas, USA, 1998.

[2] Simon Garfinkel and Gene Spafford, *Practical UNIX & Internet Security*, O’Reilly, 1996.

[3] Aleph One, ``Smashing The Stack For Fun and Profit,’’ http://www.fc.net/phrack/files/p49/p49-14.

[4] Nathan P. Smith, ``Stack Smashing Vulnerabilities in the UNIX Operating System,’’ http://reality.sgi.com/nate/machines/security/stack-smashing/.

[5] CERT, ``Multiple Buffer Overflows in Kerberos Authenticated Services,’’ http://www.cert.org/advisories/.

[6] E. Spafford, ``The Internet Worm Program: Analysis,’’ Computer Communication Review, 1989.

[7] Stevens, W. Richard, *Advanced Programming in the UNIX Environment*, Addison-Wesley, 1992.

[8] Aleph One, ``Question on Buffer Overflow,’’ http://www.securityfocus.com/.

[9] Andrew S. Tanenbaum, *Modern Operating System*, Prentice Hall, 1992.

[10] Michael Beck, Harald Bohme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, Harold and Bohme, *Linux Kernel Internals*, Addison-Wesley, 1996.

[11] Remy Card, Eric Dumas, and Franck Mevel, *The Linux Kernel Book*, Wiley, 1998.

[12] Mudge, ``How to Write Buffer Overflows,’’ http://10pht.com/advisories/bufero.html.

[13] OpenBSD, ``OpenBSD Security,’’ http://www.openbsd.org/security.html.

[14] Solar Designer, ``Non-Executable User Stack,’’ http://www.openwall.com/.

[15] Alexandre Snarskii, ``FreeBSD Insecure Library Function’s Stack Integrity Check,’’ ftp://ftp.lucky.net/pub/unix/local/libc-letter, 1997.

[16] Richard W M Jones and Paul H J Kelly, ``Backwards-compatible Bounds Checking for arrays and pointers in C programs,’’ http://www-ala.doc.ic.ac.uk/~phjk/BoundsCheckinig.html.

[17] Evan Thomas, ``Attack Class: Buffer Overflow,’’ http://students.ou.edu/W/Amos.P.Waterland-1/wellspring/buffer_overflow.html.

[18] Crispin Cowan, ``Buffer Overflow and OS/390,’’ http://geek-girl.com/bugtraq/1999_1/0481.html.

[19] Tim Newsham, ``StackGuard: Automatic Protection from Stack-smashing Attacks,’’ http://www.securityfocus.com/.

[20] Matt Conover, ``w00w00 on Heap Overflows,’’ http://www.w00w00.org/articles.html.

[21] Crispin Cowan et. al., ``StackGuard Compilser: a gcc Enhancement,’’http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/compiler.html.

[22] Ham Swap-Linux, ``Linux SuperProbe vulnerability,’’ http://www.insecure.org/sploits/linux.SuperProbe.html.

[23] Steve Summit, ``Pointers to Functions,’’ http://gsu.linux.org.tr/doc/C/c_faq/~scs/cclass/int/sx10.html.

[24] R. Sekar and P. Uppuluri, ``Synthesizing Fast Intrusion Detection/Prevention Systems from High-Level Specifications,’’ USENIX Security Symposium, 1999

[25] Wenke Lee and Sal Stolfo, `` Data Mining approaches for Intrusion Detection,’’ Proceedings of the Seventh USENIX security Symposium (SECURITY ’98), San Antonio, TX, January 1998.

[26] Fu-Hau Hsu, ``The Principle, Attack Patterns and Defense Methods of Buffer Overflow Attacks,’’ ECSL-TR-87, Computer Science Department, SUNY at Stony Brook, October 2000.