

RAD: A Compile-Time Solution to Buffer Overflow Attacks

Advisee: Fu-Hau Hsu

Advisor: Professor Tzi-Cker Chiueh

Department of Computer Science

State University of New York at Stony Brook

Abstract

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 buffer overflow attacks. Buffer overflow attacks can occur in almost any kind of programs and is one of the most common vulnerabilities that can seriously compromise the security of a system. Usually the end result of such an attack is that the attacker gains the root privilege on the victim host.

Return Address Defender (**RAD**) automatically creates a safe area to store a copy of return addresses to defend programs against buffer overflow attacks. It also automatically adds protection code into applications that it compiled. Using it to protect a program does not need to modify the source code of the program. Besides, RAD does not change the layout of stack frames, so binary code it generated is compatible with existing libraries and other object files. Programs protected by RAD only experience a 1.4x performance penalty in the worst case and will no longer be hijacked by return address attackers. Finally, when an attack is detected, RAD sends a real-time message and an email to the system administrator before it terminates the attacked program. This helps the administrator to detect the intrusion in real time and helps them catch the intruder on the spot. In this paper we present possible attack patterns of return address attacks, proposed 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 buffer overflow attacks. Anecdotal evidence shows that buffer overflow attacks have already been used to attack programs since the 1960s [18]. The most famous buffer overflow 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 system. Usually the result of such an attack is that the attacker gains the root privilege on the attacked host.

Although this 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 thus continue to generate programs with this vulnerability. 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, classified attack patterns, and their defense methods. Section 3 presents the implementation details of RAD. Section 4 describes effectiveness of RAD and its performance overheads. Section 5 presents alternative implementations of RAD and our future work. Section 6 describes related works in this field. Section 7 is the conclusion.

2 Buffer Overflow Attacks

This section describes the principle of buffer overflow attacks. Subsection 2.1 discusses how buffer overflow attacks work. Subsection 2.2 introduces possible return address attack patterns. Subsection 2.3 discusses existing defense methods against buffer overflow attacks.

2.1 Principle of Buffer Overflow Attacks

If programs don't check the size of the 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 above programming error creates the breeding ground of buffer overflow attacks.

Variables with similar 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 user 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. Applying the same method to function pointer variables[23], attackers can have their injected code executed also.

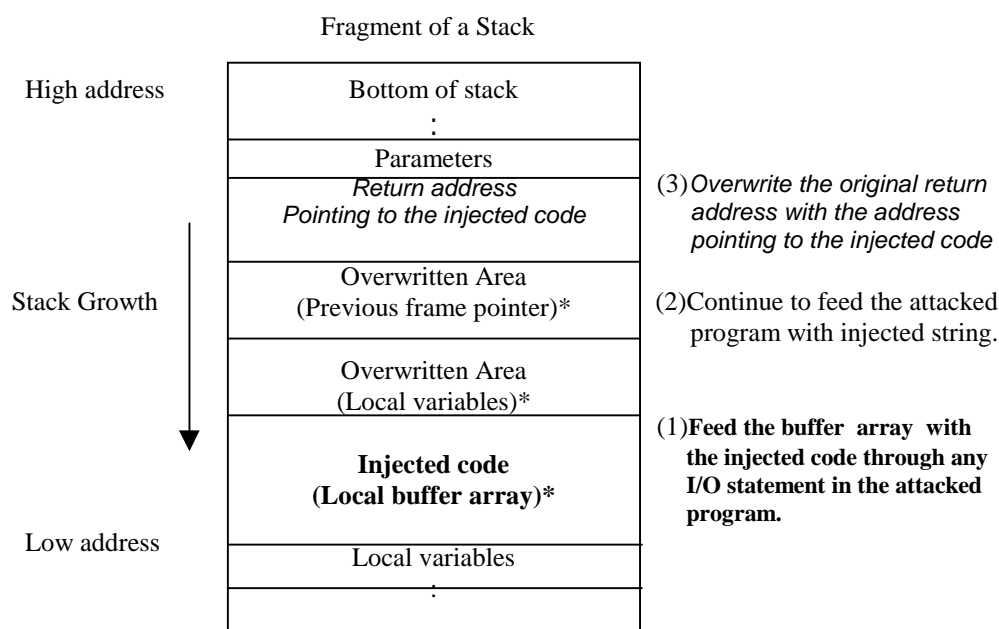


Figure 1: Buffer Overflow attack

*: Original data structure storing in this location before been overwritten by injected string .

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, see Figure 1. Most C compilers do not perform array bounds checking. As a result, arrays in C programs become the favorite targets of buffer overflow attacks. In order to launch such an attack, all an attacker needs to do is (1) compose a string containing his/her code and a return addresses pointing to the code, and (2) insert the string into the correct place in the address space of the attacked program through an I/O statement. Then when the function whose 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.

This subsection only gives a brief description of buffer overflow attacks. [3,4,20,22] give detail descriptions of this kind of attacks and several good examples.

2.2 Attack Patterns

This subsection systematically analyzes C statements that could change the contents of a memory location. Through this analysis we can explore possible return address attack patterns regardless of whether they have been implemented or not. Based on the result of this analysis, we can check the effectiveness of an anti-return address attack algorithm.

C language has 3 types of statements that could change the content of a memory location. Because of this power, these statements become the stepping-stones of buffer overflow attacks.

The 3 types of statements are as follows:

1. $A=B;$ where A and B are variables.
2. $*A=B;$ where A is a pointer variable and B is a variable.
3. $A[I]=B;$ where A[] is an array, B is a variable, and I is an integer index.

Even though there are a lot of other different C statements that could change the content of a memory location. We still can classify them into one of the above categories. e.g. $A[P[I]]=B$ belongs to category 3 and $*P[I]=A[J]$ belongs to category 2.

In the following we assume that attackers know the run-time memory layout of the attacked application very well, so whenever they inject code into memory they always know the address of the injected code.

Analysis:

1. $A=B;$ where A and B are variables.

When this statement is executed, A's address is fixed and can't be changed. Besides, the compiler doesn't use variables to store return addresses. For the above reasons, attackers can't use this statement directly to change a return address.

2. $*A=B;$ where A is a pointer variable, B is a variable.

The semantic of this statement is as follows: the memory location that A points to gets B's right value.

One attack pattern can use this statement to change a function's return address.

Attack Pattern 1:

Conditions that must exist in the attacked program for a buffer-overflow attack to succeed:

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

Under the above conditions, attackers could launch a pattern 1 attack in the following way: Before $*A=B$ is executed, use statements in conditions (1) and (2) in the attacked program to change A and B's values. Then when 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. But until now, we have not yet found any exploit code that is based on this attack pattern.

3. $A[I]=B;$ where $A[]$ is an array and B is a variable. I is an integer index.

The semantic of this statement is as follows: The I th element of array $A[]$ gets B 's right value.

Two attack patterns can use this statement to change a function's return address.

Attack Pattern 2:

Conditions that must be satisfied in the attacked program:

- (1) A loop statement that copies user input into a buffer array without checking its bounds, and the array is adjacent to variable B .
- (2) A loop statement that copies user input into array A without checking array A 's bounds. Each iteration of the loop statement increases index I by one. Array A is adjacent to a return address.

Under the above conditions, attackers could launch a pattern 2 attack in the following way: Use statement in (1) to set B 's value. Then use statements in condition (2) to input B 's new value into array A and repeat doing so until the return address is overwritten by the new value. This attack pattern is the most common method used in buffer overflow attacks. All examples in [3,4] use this attack pattern. In this case, not only the return address, but also the whole memory area between array A and the return address are modified.

Attack Pattern 3:

This attack pattern is similar to attack pattern 1.

Assume $A[k]$ holds a function's return address. If we could use methods in attack pattern 1 to overflow the content of index I with k , then we can change the return address to any value we want.

The above analysis shows that if an attacker could inject his/her code into memory and use any of the above attack patterns to change a return address to point to the inserted code, then he/she can have the injected code executed with the attacked program's privilege.

Knowing all possible attack patterns, we can use them to evaluate the correctness of an anti-return address attack algorithm. For instance, OGI's [1] algorithm can effectively deal with pattern 2's attack, which is the most common form of buffer overflow attacks. But because pattern 1 and pattern 3 attacks can change the content of any memory location within a user address space, the OGI algorithm can not survive these attacks. However no known exploit code use these two attack patterns.

2.3 Defense Methods

Injecting malicious code and addresses into a victim program (step1), changing its control flow at run time (step 2), and executing the injected code (step 3) are the 3 essential steps to launch a buffer overflow attack. A successful attack must have all of these 3 steps. Several different solutions have been proposed or implemented to handle the buffer overflow problem. Through preventing the accomplishment of one of more of these 3 steps, these approaches defend programs against the buffer overflow attacks. According to the strategy they use, we can classify these defense methods into the following three categories.

2.3.1 Defense methods that don't allow buffer overflow at all:

This kind 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 Richard Jones and Paul Kelly's case, they developed an extension to the GNU C compiler to automatically perform array bounds and pointer checking. In OpenBSD's case, they manually modify the source code

to perform array bounds and pointers checking.

2.3.2 Defense methods that allow buffer overflow but don't allow unauthorized change of control flow:

This kind of methods allows foreign code to be injected, but prohibits unauthorized changes of control flow. So attackers can inject their code into memory and can change some addresses, but control flow will not be transferred to the injected code. RAD and OGI's StackGuard[1] use this strategy to protect programs. RAD uses RAR to prevent injected address from being used as return points of function calls. StackGuard uses canary words to achieve the same goal.

2.3.3 Defense methods that allows change of control flow, but prevent the execution of sensitive code:

These methods allow step 1 and 2 to occur, but disable step 3. So code and addresses can be injected into memory and control flow can be transferred to the injected code, but the injected code can not be executed completely. Solar Designer's non-executable stack [14], intrusion detection, e.g., R. Sekar's intrusion detection [24] and Wenke Lee's [25] intrusion detection, and **SCD** (System Call Defender) use this strategy to defend programs against 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, but the code can not be executed. In R. Sekar's method, before a program is executed, they manually build normal/abnormal behavior patterns in terms of system calls and their arguments for each program to be defended. By comparing the run-time behavior of the program with these patterns, they detect/prevent the attacks. Wenke Lee's method uses the similar principle, but they use data mining approach, some sort of neural-network learning from training sets, to build up the patterns. Both intrusion detection methods allow injected code to be executed, but by using their approaches to detect abnormal behavior of a program or known intrusion patterns they prevent the execution of sensitive code.

SCD is another anti-buffer overflow attack project developed by us, ECSL Lab. of Computer Science Department of SUNY at Stony Brook. The principle of SCD is as follows: In order to compromise a computer system, attackers' injected code must be able to promote or change attackers' privileges. System calls, such as `exec()`, are powerful and convenient tools to achieve these goals. SCD is a patch to the Linux kernel that allows step 1, 2 and even 3 to occur. But by prohibiting the execution of illegal system calls, SCD defends programs against buffer overflow attacks. In Linux, when programs make a system call the return address of it will be saved in the kernel stack. From the above information, we can know from where this system call originates. Except the signal-handler-return system call, no system call will be issued from the stack segment. By checking the segment that a system call originates from, SCD can decide whether the system call is legal or not. We have implemented a SCD prototype. The challenge we encountered is that if in their exploit code attackers don't enter a system call directly, but only set up parameters needed in that system call and use system calls in the original attacked program to mount an attack. For instance, the injected code prepares the parameters of a system call and then jumps to an "int 80H" instruction to actually execute the system call. The current SCD version can not detect such an attack.

3. Return Address Defender: Protecting Return Addresses from Being Used to Execute Inserted Code

Return address defender is a patch to gcc-2.95.2 which automatically adds protection code into the function prologues and epilogues of the programs that it compiles. So when we use it to protect an application, there is no need to make any modification to the source code.

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 return addresses in the data segment called Return Address Repository (**RAR**). By setting neighbor regions around RAR as read-only, we can defend RAR against any attempt to modify it through overflowing. After RAR's security is guaranteed, each time when a return address in a stack

frame is used to jump back to the caller function, this address is checked with the addresses in RAR. A return address will be treated as a safe address to use only if RAR also contains the same address.

RAD provides two ways, MineZone RAD and Read-Only RAR, to protect the return addresses in RAR. Both methods are portable. MineZone RAD is more efficient while Read-Only RAR is more secure. Subsection 3.1 describes MineZone RAD. Subsection 3.2 describes Read-Only RAR. Subsection 3.3 describes additional buffer overflow-related security problems due to `setjmp()` and `longjmp()` system calls. Subsection 3.4 proves the correctness of RAD.

3.1 MineZone RAD: Setting Up a Mine Zone in Both Sides of RAR

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.

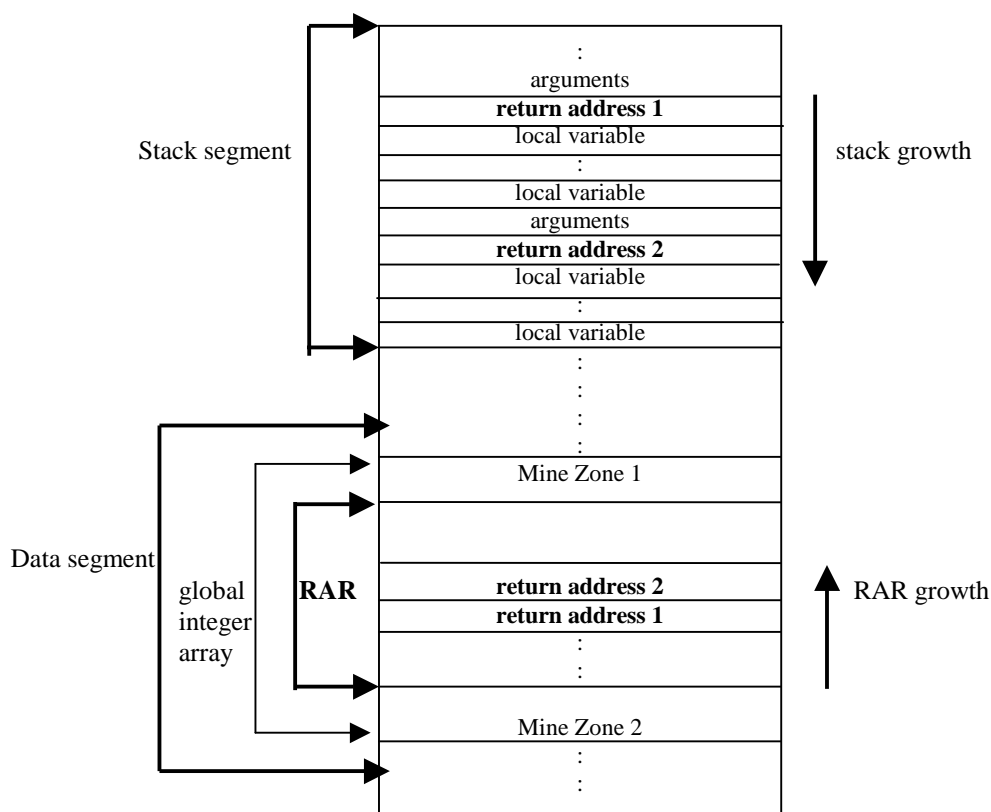


Figure 2: RAR and Mine Zones

```

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: Pushing Current Return Address into RAR

The middle part of the global integer array is RAR which is an integer array. Each element of it holds a return address. The first and third parts, call mine zones, are set as read-only areas by mprotect() system call so any writing into these areas will cause a trap. All security functionalities are implemented by instructions added in the new function prologues and epilogues but the stack frame layout of each function is still the same as the traditional one, so 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 will compare the current return address with the top return address in RAR. If they are the same, then RAD will pop it from RAR and go back to the caller. Otherwise this means someone is attacking the return address. (See subsection 3.3 for an exception.) A real-time message and an email are sent to the system administrator and the attacked program is terminated. In figure 3 and figure 4 we list the pseudo code of these additional instructions.

```
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: Comparing Return Addresses

MineZone RAD is an efficient algorithm and it can survive the second attack pattern, which is also the most common form of buffer overflow attacks. But this method can't survive attack pattern 1 and pattern 3. If there is any such exploit code, they must be very rare, because in order to launch such an attack all conditions for these patterns must be satisfied in the attacked program simultaneously. In the next subsection we present a second method which resists all attack patterns.

3.2 Read-Only RAR: Setting RAR as Read-Only

```
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: Pushing Current Return Address into RAR

Read-Only RAR is similar to Mine-Zone RAD. But instead of setting up mine zones, Read-Only RAR

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. In figure 5 and figure 6 we list their pseudo code.

In Read-Only RAR, RAR is set as read-only most of the time. The only time that it could be written is in the function prologues when the current return address is pushed into RAR. But there is no other I/O in these function prologue instructions, so pattern 1 and pattern 3 attacks don't have any chance to update RAR. Of course, because RAR is set as read-only updating it in function prologues requires adding two extra system calls to each function call, a serious performance penalty.

```
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: Comparing Return Addresses

3.3 Problems Caused by Setjmp() and Longjmp()

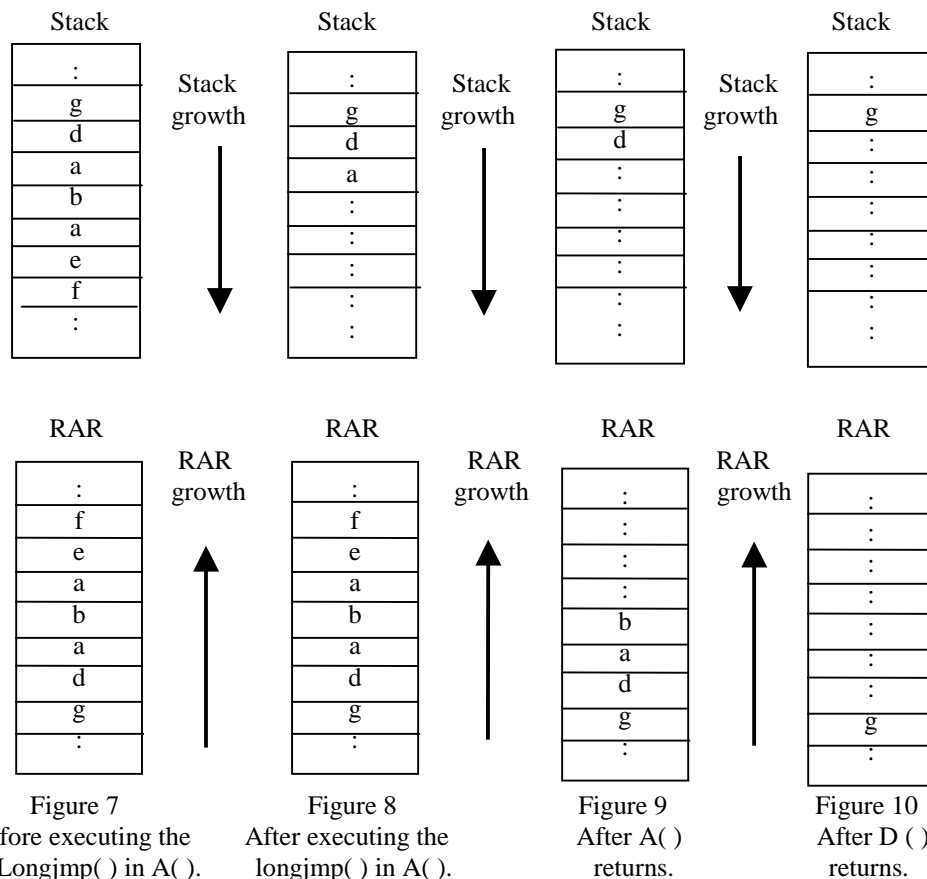
In previous subsections, we describe the design of RAD, with the following assumption. When a function is called, its stack frame is pushed into the stack and is not popped from the stack until it finishes and returns. And when a function returns, only its stack frame is popped.

But the assumption is not always true for C programs. System calls `setjmp()` and `longjmp()` [7] allow users to bypass several functions in the call path of the current function to directly jump back to the function executing `setjmp()`. Users use `setjmp()` to set the return location and use `longjmp()` in a different function to go back to the `setjmp()` statement. If between the execution 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 top stack frame don't match in this case. According to 3.1 and 3.2, if the mismatch occurs, RAD will treat this as an attack and terminate the program.

Now that executing `longjmp()` will pop more than one stack frame, we can address this problem by simulating the above action by popping the return addresses in RAR accordingly. When finding 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 there is no match when RAR is empty, then it means someone is launching an attack, otherwise the return address is safe to use.

However this scheme leads to another problem. We use figures from Figure 7 to Figure 10 to illustrate this problem. 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 shows a particular calling sequence, the 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 as Figure 8. When function A returns, the stack layout is as Figure 9. But using the new method, RAD only pops the return addresses of F, E and A, as shown in Figure 9. Obviously the two layouts are differently. Will this 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 10. At this moment RAD will pop return addresses b, a, and d. So now they become the same again.



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 to execute according to the program its own execution plan. So the layouts of return addresses in RAR and in the stack do not need to be the same. RAD only need to guarantee that no illegal entry is added into RAR, (RAD uses MineZone RAD and Read-Only RAR to achieve this goal.) , and when a legal entry is added into RAR, the entry is still in RAR when the function, whose function prologue pushed the return address into RAR, prepares to return, (The following subsection will prove this.).

3.4 Proof of Correctness of RAD:

What follows are notations used in the proof:

- $A_n()$: the nth instance of function A().
- $SF-A_n$: the stack frame of $A_n()$.
- $RA-A_n$: the return address of $A_n()$.

Definition:

execution sequence:

A sequence of functions which are chained together through function calls or longjmp(s).

setjmp-longjmp execution sequence from A_n() to B_m():

A sequence of functions, starting from A_n() (A_n() executes setjmp().), including all functions whose stack frames are between A_n()'s and B_m()'s stack frames in the stack when B_m() is executing longjmp(), and ending at A_n() . (When B_m() executes longjmp(), the control flow is transferred back to A_n() .)

For example:

If after A() executes setjmp(), A() calls function B() which in turn calls function C(). C () then calls D() which executes longjmp() to go back to A(). Then A-B-C-D-A is a setjmp-longjmp execution sequence from A() to B().

generic execution sequence:

A sequence of functions that are chained together through function calls, but none of those functions executes longjmp() and the last function called does not call other functions when it is active.

For example:

A(), B(), C(), and D() are functions. A() calls B() which in turn calls C(). C() then calls D(). D() doesn't call other functions when D() is active. Then A-B-C-D-C-B-A is a generic execution sequence.

execution tree:

A tree that describes the execution of a program and the transfer of the control flow between functions. The node of the tree represents an instance of a function. The edge of the tree represents the transfer of control flow from one instance of a function to another instance of a function through a function call or longjmp(). The root of the tree represents the main() function. If function A() uses a function call or a longjmp() statement to transfer control flow to function B(), then there will be an edge starting from A()'s node and pointing to B()'s node. Figure 11 is an execution tree example.

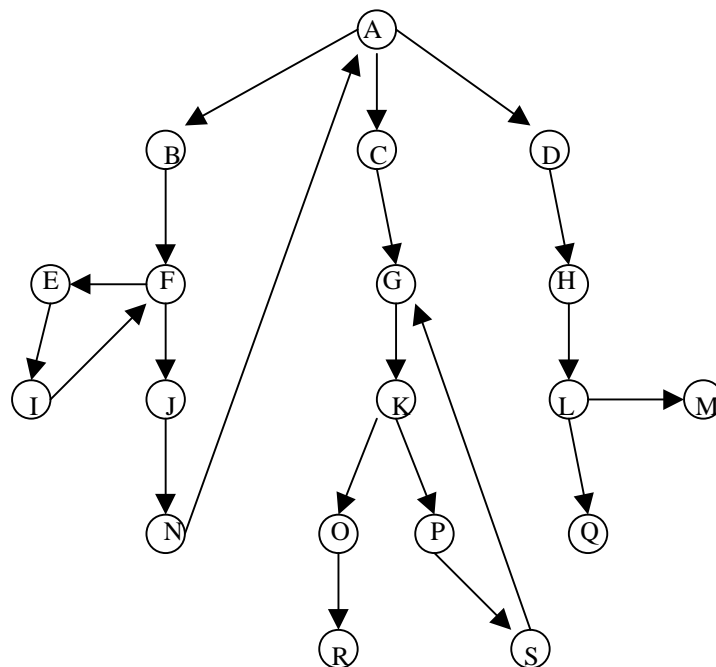


Figure 11: An Execution Tree.

According to the definitions, in Figure 11 we can find the follows sequences.
setjmp-longjmp execution sequence from A to N: A-B-F-J-N-A

setjmp-longjmp execution sequence from G to S : G-K-P-S-G.

setjmp-longjmp execution sequence from F to I : F-E-I-F

generic execution sequence: A-D-H-L-M-L-H-D-A, L-Q-L, and K-O-R-O-K

execution sequence A-C-G-K-P-S-G-C-A is an execution sequence containing a setjmp-longjmp execution subsequence.

Obviously, setjmp-longjmp execution sequences describe those paths that form loops in the execution trees. Generic execution sequence describe those paths that form lines ending at leaf nodes in the executions trees

In the following analysis, $B_m()$ represents the function currently executed. According to its property we can classify $B_m()$ into one of the following 4 classes:

Class 1:

“ $B_m()$ executes `longjmp()` and after the function is executed, execution is transferred from $B_m()$ to $A_n()$. Besides, this setjmp-longjmp execution sequence from $A_n()$ to $B_m()$ is the only setjmp-longjmp execution sequence executed during the time that $A_n()$ is active.”

`longjmp()` causes the popping of N ($N \geq 0$) stack frame, N is the number of stack frames above $A_n()$'s stack frame when $B_m()$ is executing `longjmp()`, but leaves RAR unchanged. So the setjmp-longjmp execution sequence from $A_n()$ to $B_m()$ results in that RAR is added N additional entries, but restores the stack to the state when `setjmp()` in $A_n()$ is executed, i.e. $SF-A_n$ is the topmost stack frame.

Because the setjmp-longjmp execution sequence from $A_n()$ to $B_m()$ is the only setjmp-longjmp execution sequence executed during the time $A_n()$ is active, only functions without executing `longjmp()` are executed, if there is any such function. The execution of these functions forms several generic execution sequences. Functions in a generic execution sequence push their return addresses into RAR when they are entered and pop their return addresses from RAR when they return. This just simulates stack's behavior. So the execution of a generic sequence doesn't change the layouts of RAR and the stack when they finish.

The above analysis shows that neither kinds of the above execution sequences delete $A_n()$'s return address from RAR. Due to the above reasons, right before $A_n()$ returns, the N additional entries are still in RAR and are the topmost N entries. $A_n()$'s return address, $RA-A_n$ is still in RAR and is right below the above N entries.

Assume $A_n()$ returns to $D_r()$. When $A_n()$ returns, RAD begins to pop return addresses from RAR. This behavior may pop $N+1$ entries, more than $N+1$ entries, or less than $N+1$ entries from RAR. But when RAD find that the topmost entries in RAR is $RA-A_n$, it stops popping after $RA-A_n$ is popped. So it is impossible for RAD to pop more than $N+1$ entries from RAR in this case. But RAD may pop $N+1$ entries or less than $N+1$ entries from RAR. In the former case, no more or less return addresses are popped, so obviously no problem occurs. If RAD pops fewer return addresses from RAR, RAD still can be sure that the return address is safe to use. Popping fewer return addresses means more return address information is kept. So no error occurs also.

The above analysis shows that if a class 1 function is the function currently executed, then when $SF-A_n$ is popped, RAR has either the same return addresses as or more return addresses than what the stack has. No return address is lost.

Class 2:

“ $B_m()$ does not execute `longjmp()` and during the time that $B_m()$ is active no `longjmp()` is executed.”

In this case only functions without executing `longjmp()` are executed. The execution of these functions forms several generic execution sequences. The analysis in class 1 shows when $B_m()$ returns, no return address is lost.

Class 3:

“ $B_m()$ executes the `longjmp()` and when the `longjmp()` is executed, execution is transferred from $B_m()$ to $A_n()$. Besides instead of the `setjmp-longjmp` execution sequence from $A_n()$ to $B_m()$, there are other `setjmp-longjmp` execution sequences executed during the time that $A_n()$ is active.”

In this case, during the time that $A_n()$ is active, there are several `setjmp-longjmp` execution sequences. Generic execution sequences may also happen, but when these generic execution sequences finish, RAR and the stack remain unchanged, so we do not need to consider them here.

Class 1 shows that each `setjmp-longjmp` execution sequence makes RAR have more entries than the stack and when the function containing a corresponding `setjmp()` statement returns, fewer or the same return addresses are popped from the RAR. This makes RAR have either the same return addresses as or more return addresses than the return addresses that the stack has. From above analysis, we can conclude that each execution sequence containing a `setjmp-longjmp` execution subsequence will make RAR have either the same entries as or more entries than the stack and all these entries are added on top of $A_n()$'s return address. So when all execution sequences finish and $A_n()$ returns, RAR has either the same return address entries as or more return address entries than the return addresses the stack has. In this case, no return address is lost also.

Class 4:

“ $B_m()$ doesn't execute `longjmp()` directly. But during the time that $B_m()$ is active, some `setjmp-longjmp` execution sequences are executed.”

Based on the results from Class 1 to Class 3, we know right before $B_m()$ returns, RAR has either the same return addresses as or more return addresses than the return addresses that the stack has and $B_m()$'s return address is right below these additional addresses. So when $B_m()$ returns, RAR still has either the same return addresses as or more return addresses than the return addresses that the stack has.

Analyses from class 1 to class 4 show that no matter what kind of functions the currently executed function is, when it returns RAR has either the same return addresses as or more return addresses than the return addresses that the stack has. So no return address is lost. RAR works correctly.

Keeping more return addresses in RAR means popping fewer return addresses from RAR. It also means that RAD performs less computation and the performance overhead is reduced.

The test program `ctags` in section 4 uses `setjmp()` and `longjmp()` system calls. When being fed with the same input, both the RAD-compiled `ctags` and the GCC-compiled `ctags` generate the same results. So RAD correctly processes `setjmp()` and `longjmp()`.

From the above analyses and experiment, we can conclude that the algorithm used to deal with problems caused by `setjmp()` and `longjmp` is correct.

4. Effectiveness and Performance

Section 4 discusses the effectiveness of RAD in defending against buffer overflow attacks and describes experimental results of the RAD prototype. Subsection 4.1 talks about other implementation-related issues. Subsection 4.2 discusses the effectiveness of RAD. Subsection 4.3 describes the micro-benchmark and macro-benchmark of RAD.

4.1 Other Implementation-Related Issues

The code implementing RAD consists of two parts, patch code in GCC and code 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 the 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 area of the address space that `mprotect()` uses to protect RAR. So when a memory access violation occurs, 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 in the user address space before executing `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

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 are only small modifications of 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 resists the attack effectively. Originally, our next step is to find more exploit codes and then test them against their target programs with or without the protection of RAD. But for the following reasons, we chose a different way to prove the effectiveness of RAD. Currently those who attack return addresses only know that stack return addresses are their targets. They don't know in RAD there is another copy in data segment. This is a new feature added by RAD, so even though we could show that RAD can deal with existing exploit codes, we still can not be sure that RAD can protect return addresses well, because none of them know the existence of the RAR copy, let alone attack it. So instead we use attack patterns to evaluate the effectiveness of RAD, because attack patterns can test all possible attack patterns, whether they are implemented or not.

Effectiveness of MineZone RAD:

(1) Attack pattern 2 is the most common form of buffer overflow attacks. A loop statement in the attacked program that reads input from outside and doesn't perform array bounds checking can easily be used as a target of attack. In order to launch a pattern 2 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 changed. In MineZone RAD, both sides of RAR have a one page read-only mine zone to protect it. Any one trying to use pattern 2 to change an 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 2 attack successfully. Aleph One's exploit code belongs to this pattern.

(2) If a program satisfies all the conditions listed in attack pattern 1 and 3, then even though it is protected by MineZone RAD, it may still be hijacked by attackers who use attack pattern 1 or 3 to attack the program. We think exploit code based on these attack patterns is rare, because only attacked programs that satisfied all the conditions described earlier simultaneously have this vulnerability. In fact, we have not yet found any exploit code based on these attack patterns.

Effectiveness of Read-Only RAR:

In Read-Only RAR, except in function prologues, RAR is read-only and in function prologues there is

no any I/O statement and loop statement, so attackers do not have any chance to change RAR. So Read-Only RAR defends return addresses against all attack patterns.

4.3 Performance Cost

This section describes tests that we use to evaluate the performance overhead of RAD. OGI[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 RAD. Subsection 4.3.1 presents the micro-benchmark results for MineZone RAD and Read-Only RAR. Subsection 4.3.2 presents the macro-benchmark results for MineZone RAD and Read-Only RAR.

4.3.1 RAD Micro-Benchmark

We wrote 3 C programs to discover 3 kinds of additional performance cost imposed by RAD. We use weight to measure the penalty which is the ratio of a function's additional performance cost associated with RAD to the function's original performance overhead.

$$\text{weight} = \frac{\text{a function's additional performance cost associated with RAD}}{\text{the function's original performance overhead}}$$

Each program has only two functions. Except the main function, there is another function which increases the value of a variable by one. The function prototypes of these functions are different. We call the function 100000000 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 MB main memory.

In program 1, we call a function, void inc(), 100000000 times to increment a global variable 100000000 times. The function has no arguments and no return value.

In program 2, we call a function, void inc(int *), 100000000 times to increment a local variable of main() 100000000 times. The function has no return value, but has one argument which 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 100000000 times to measure RAD's penalty. The function has a return value and an argument.

Table 1 and Table 2 shows the micro-benchmark results of MineZone RAD and Read-Only RAR

Function Prototype	Original run-time	MineZone RAD run-time	weight
void inc()	12814378	30897126	1.41
void inc(int *)	17334197	35418721	1.043
Int inc(int)	18089581	36924768	1.041

Table 1: Micro-benchmark results of MineZone RAD

Function Prototype	Original run-time	Read-Only RAR run-time	weight
Void inc()	12814378	2696119743	209.40
Void inc(int *)	17334197	2628705475	150.65
Int inc(int)	18089581	2651345171	145.57

Table 2: Micro-benchmark results of Read-Only RAR

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 overhead 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 RAR, not only the above protection operation needs to 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 RAR is higher.

4.3.2 RAD Macro-Benchmark:

In subsection 4.3.1, we measure performance penalties on functions containing only one statement. They provide upper bounds 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 programs, `ctags` and `gcc`.

`Ctags` is a UNIX command which generates an index file for several languages, e.g. C. The version we used is `ctags-3.2`. Totally there are 9488 lines of source code. The program that `gcc` and RAD-protected `gcc` compile is a dictionary proxy server with 4500 lines of source code.

For each program, we use "time command parameters" to measure the execution time of both the original program and the RAD-protected version. From these measurements, we calculated the performance penalty, see Table 3 and Table 4.

Size of argument file	Program tested	User time	System time	Real time
11991 lines	Original <code>ctags</code>	0.57	0.05	0.62
	MineZone RAD-protected <code>ctags</code>	0.58	0.05	0.63
	Read-Only RAR-protected <code>ctags</code>	8.16	19.17	27.32

Table 3: Macro-benchmark results of `ctags`

Size of argument file	Program tested	User time	System time	Real time
4500 lines	Original <code>gcc</code>	3.53	0.19	3.72
	Mine Zone RAD-protected <code>gcc</code>	4.67	0.2	4.87
	Read-Only RAR-protected <code>gcc</code>	20.46	50.43	70.89

Table 4: Macro-benchmark results of `gcc`

As mentioned in previous subsection, the additional cost of a RAD-protected program comes from the additional instructions added in 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 RAR-protected `gcc`, 18x, is smaller than the penalty of Read-Only RAR-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 Alternative Methods and Future Work

In this section we discuss other implementation alternatives related to RAD. Subsection 5.1 discusses alternative implementation methods of RAD. Subsection 5.2 talks about future enhancements to RAD.

5.1 Alternative implementations

Saving a copy of return addresses in a secure place and checking them when functions return is the principle of RAD. Based on this principle, there are several different implementations. But different implementations have different tradeoff among security, performance, and memory usage. Subsection 5.1.1 talks about an alternative implementation of MineZone RAD. Subsection 5.1.2 describes an alternative implementation of Read-Only RAR.

5.1.1 Replacing Mine Zones with Debug Registers

In MineZone RAD, we mark the two pages adjacent to RAR as read-only to protect RAR. Pentium Processors have four debug registers which when loaded can monitor read, write, and execute access to specific addresses. So we can replace the mine zones with two debug registers that monitor the two ends of RAR. But not every processor has debug registers, so this implementation is an architecture-specific solution and thus non-portable.

5.1.2 Saving Return Addresses in Process Control Table

Creating a safe place to store RAR is a key element of RAD. The kernel space is obviously a safe place. Linux uses buddy system [9,10,11] to allocate kernel memory resource. Even though a process's process control table only consumes about 1000 bytes, in our case 960 bytes, Linux still allocates 2 pages (8K) for it. So we have about 7KB free memory left, usually this is enough to hold RAR. The major performance penalty of Read-Only RAR comes from the `mprotect()` system call. If we could build high-weight system calls to save and retrieve return addresses from process control table, we can decrease RAD's penalty. But this means we need to modify the kernel and if we need to save more than 7k return addresses, we need to consume valuable kernel memory resource for each process.

5.2 Future Work

In this subsection we discuss future work to enhance RAD. Subsection 5.2.1 talks about using the compiler to combine MineZone RAD and Read-Only RAR. Subsection 5.2.2 discusses ways to catch the intruders. Subsection 5.2.3 talks about function pointers and binary RAD.

5.2.1 Using Compilers to Combine MineZone RAD and Read-Only RAR

From subsection 2.2 we know that, in order to launch an attack, certain conditions must be satisfied in the attacked program. For all of the attacked patterns, they all need loop statements in the attacked programs. These loop statements continually get or copy user input into a buffer array without checking array bounds. If a function does not have such dangerous statements, there is no need to protect the return address in its stack frame. Even if a function does have such a dangerous statement, but does not have other statements needed to launch a pattern 1 or pattern 3 attack, MineZone RAD is secure enough. Only when all the dangerous statements appear in the same attacked programs in the right orders is Read-Only RAR needed. So if we can use compilers to detect those dangerous statements in each function and choose an appropriate method, we can develop a more efficient and secure RAD.

5.2.2 Catching the Intruders

When RAD detects an attack, it sends a real-time message and an email to the system administrator and then terminates the attacked program. After that the system administrator can either use a watch-dog process or an `inetd` daemon to restart the program. But when RAD detects an attack, the chance that the intruder is still on-line is very high. Before RAD terminates the attacked program, the intruder still doesn't know that his/her attack has already been detected, so if we can use this opportunity to backtrack the source of intrusion, maybe we can catch them on the spot. Or we can create a forged shell to record all commands the intruder typed to see what the intruder intends to do.

5.2.3 Function Pointers and Binary RAD

Function pointer variables are also targets of buffer overflow attacks. By overwriting the content of a function pointer variable with the address of the injected code, attackers can have their injected code executed with the attacked program's privilege. Exploit code in [22] uses this scheme to attack the `suid` root program `SuperProbe` to get the root privilege. Currently RAD only protects the stack return addresses. So expanding the protection scope to cover function pointer variable will be the focus of enhancements to RAD. This involves inserting a check to each function call instruction.

Finally, the input of current RAD version is the source code, in the future, we plan to modify it so that the new RAD could work on binary files directly.

6 Related Work

In this section, we review other solutions to buffer overflow attacks and analyze these works' tradeoff among performance penalty, security strength, and portability.

6.1 OGI StackGuard

Crispin Cowan and Calton Pu et. al. [1] have developed a gcc patch StackGuard which protects return addresses from being modified to point to the injected code. There are 3 variants of StackGuard, Canary version, MemGuard Register, and MemGuard VM. StackGuard provides an adaptive response to intrusions so it will automatically switch between the above variants when attacks are detected StackGuard does not need to make any modification in the source code and the binary code generated by StackGuard is compatible with existing libraries.

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 2 attacks with little performance penalty. But if attackers can guess the canary value, this method will not work. Besides under certain conditions attackers could overwrite the precomputed canary values with their own values to turn off the protection [19]. Due to alignment requirement, it is possible to overwrite a return address but skip over the canary word. Besides, canary version can't survive pattern 1 attacks and pattern 3 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. Finally, canary version will change 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. For instance, `gdb` inspects other program's stack frames, and thus will fail to produce correct stack traces when applied to StackGuard-protected programs. The Linux kernel also does not compile under StackGuard. Thus kernels, and programs to be debugged with `gdb` should be compiled without StackGuard protection."

Both the 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. And "by installing a trap handler, they catch writes to protected pages, and emulates the writes to non-protected words on protected pages."

These two versions are more secure than the canary version. But not every architecture has debug registers. And 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 sets as read-only to reduce the performance penalty, then return addresses in these stack frames are without protection and vulnerable to return address attacks. For example, `A()` and `B()` are two functions. For a special calling sequence, `B()` is executed after `A()` and `B()`'s return address is protected by a debug register but `A()`'s return address isn't. Besides, `A()`'s and `B()`'s stack frames are on the same stack page. If the address of one of `A()`'s local buffer arrays is passed to `B()` as a parameter and there are dangerous statements which copy user input into this array without checking its bounds, then attackers can use buffer overflow attacks to

change the return address and inject code. When A() returns, the injected code will be executed. MemGuard VM version has a very large performance penalty due to frequent memory references to a read-only area. All the 3 variants of StackGuard can not protect function pointers from being attacked.

6.2 Intrusion Detection

Intrusion detection is designed to protect computer systems from being compromised by intruders no matter what kind of attacks is utilized to launch the attacks. So it could prevent buffer overflow attacks also. This approach builds normal/abnormal behavior patterns for each program or resources, e.g. user accounts and system kernel, to be defended. By comparing the run-time behavior of each process or resources to be protected, this method detects intrusions.

According to following observation about attacks, “Regardless of the nature of an attack, damage can ultimately be caused only via system calls made by processes running on the attacked host. It is hence possible to prevent damage due to attacks if we can monitor every system call made by every process, and prevent damage-causing calls from being executed”, R. Seker et al [24] define 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 document using a core language, called Regular Expression for Events (REE). So correctly using ERR to accurately describe the program behavior is important for this method to work.

Wenke Lee et al [25] use data mining approach, some sort of neural-network learning from training sets, on audit data to discover the patterns of resources to be protected. So before the patterns is established large amount of data must be collected for the intrusion detection system to learn.

Both strategies' effectiveness relies on the existence of the correct patterns of programs. So whenever we need to execute a program protected by this approach, not only the executable file must be available but also the pattern must be available. Besides any change on the source code must have a corresponding change on the patterns.

6.3 Insecure Library Function's Stack Integrity Check

Alexandre Snarskii has written a patch to libc to make FreeBSD unexploitable 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.

6.4 Bounds Checking for Arrays and Pointers in C Programs

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 checked code is compatible with unchecked code. In their approach, every pointer value is valid for only one memory region which 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 objects other than return addresses, e.g. function pointers. But the performance penalties are substantial. For an ijk matrix multiplication, there is a 30x slowdown. Moreover, in their implementation, under certain conditions bounds checking is disabled. According to their paper, “a pointer to an array passed as a parameter can be increased to point to the next parameter without an error being reported.” So if the next parameter is a function pointer, then theoretically it is possible to inject code into the array and change the function pointer to point to the injected code to launch a buffer overflow attack.

6.5 Solar Designer's Non-Executable Stack

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

Linux single handler returns need an executable stack. Nested function calls and trampoline functions also need an executable stack to work properly. And 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 for attackers to launch a buffer overflow attack.

6.6 OpenBSD Modification of Source Code

Because programs without bounds checking are the sources of buffer overflow attacks, the most straightforward way to seal this security breach is to modify the source code. OpenBSD [13] improves system security in this way. "The process we follow to increase security is simply a comprehensive file-by-file analysis of every critical software component", they said. Using this strategy, they fixed lots of security holes and created more robust and efficient code. But manually modifying thousands of lines of source code is not an easy work.

7 Conclusion

We have shown that RAD could effectively defeat return address attack, the most common form of buffer overflow attacks. We also analyze the effectiveness and performance penalty of RAD, which shows the performance penalty imposed by RAD is modest. Binary code generated by RAD is compatible with existing libraries and other object files. When we use RAD 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. Through the above efforts we believe RAD provides an effective way to protect computer systems against buffer overflow attacks.

References

- [1] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle and Qian Zhang. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks", Proceedings of the 7th USENIX Security Conference, San Antonio, Texas, USA, 1998
- [2] Simson 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, the Computer Emergency Response Team Coordination Center. <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", Bugtraq Mailing List. <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 Internel*, 1996.
- [11] Remy Card, Eric Dumas, and Franck Mevel. *The Linux Kernel Book*, 1998.
- [12] "Mudge". *How to Write Buffer Overflows*. <http://10pht.com/advisories/bufero.html>.
- [13] 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. *Posting to Bugtraq Mailing List*. http://geek-girl.com/bugtraq/1999_1/0481.html
- [19] Tim Newsham. *Posting to Bugtraq Mailing List*. <http://www.securityfocus.com/templates/archive.pike?list=1&date=1997-12-15&msg=m0xjEHc-001100C@malasada.lava.net>
- [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 Intrusin Detection", *Proceedings of the Seventh USENIX security Symposium(SECURITY '98)*, San Antonio, TX, January 1998