Methods of Protecting the Stack Overflow Vulnerability

Z. Muzaffar A. Rasheed K. Salah
Department of Information and Computer Science
King Fahd University of Petroleum and Minerals
Dhahran 31261, Saudi Arabia
Email: {zeeshanm,aimanr,salah}@ccse.kfupm.edu.sa

Abstract

With the need of sophisticated software systems, the size and complexity is also growing consequently increasing the number of vulnerabilities. Most of the vulnerabilities are related to security issues. The most dominant of such vulnerabilities are buffer overflows. Buffer overflow vulnerabilities dominate the area of remote network penetration vulnerabilities, where an anonymous network user tries to get control of host. In this paper we survey various forms of buffer overflow vulnerabilities and their solutions. Taxonomy of solutions is compiled based upon different classes of buffer overflow attacks. Further a comprehensive analysis and evaluation of each technique is carried out based on various criteria. Our analysis and evaluation can help security researchers and designer to choose among these existing solutions. In addition we propose potential enhancements to some existing solutions.

KEYWORDS: Buffer Overflow Vulnerabilities, Stack Smashing Attacks, Network Security, Operating Systems, Qualitative Analysis.

1. Introduction

Most of the today's applications are designed by keeping in mind the importance of time and space efficiency, and quite often security is not trepidation. The security of application is now becoming a major concern due to the fact that as the size and complexity of applications are growing, the numbers of bugs are also increasing resulting into the exploitation and vulnerabilities of applications.

These vulnerabilities arose due to wide usage of C/C++ language in critical software including almost all the dominating operating systems of the present time. Actually these programs are optimized for high performance and offer little error checking capability, and hence open the doors for the attackers to exploit this deficiency. The common practice using these languages is to make excessive use of arrays, pointers and memory allocation/de-allocation, on which the programmers don't have any control during program execution. There are various C/C++ library routines that perform different sort of manipulations using these arrays and pointes etc. For example, the *strcpy(char * dst, const char *src)* function copies a string pointed to by *src* to the string pointed to by *dst*. Now the problem is that this built-in C function doesn't perform the bound checking for the

dst while copying from the *src* [12,16,20]. It's the responsibility of the programmers to ensure whether the destination buffer has ample memory to completely accommodate the source string. Unfortunately, these checks are often omitted in existing software (including well-known modern operating systems). Consequently, the missing checks could be exploited by the attackers to gain the control of a computer by launching buffer overflow attacks.

Buffer overflows have been the most common form of security vulnerabilities in the last 14 years [25] (see Figure 1). Moreover, buffer overflow vulnerabilities dominate the area of remote network penetration vulnerabilities; where an anonymous attacker seeks to acquire partial or full control of the targeted host. Because these vulnerabilities enable anyone to take the control of the host, they represent the one of the most serious security threats nowadays.

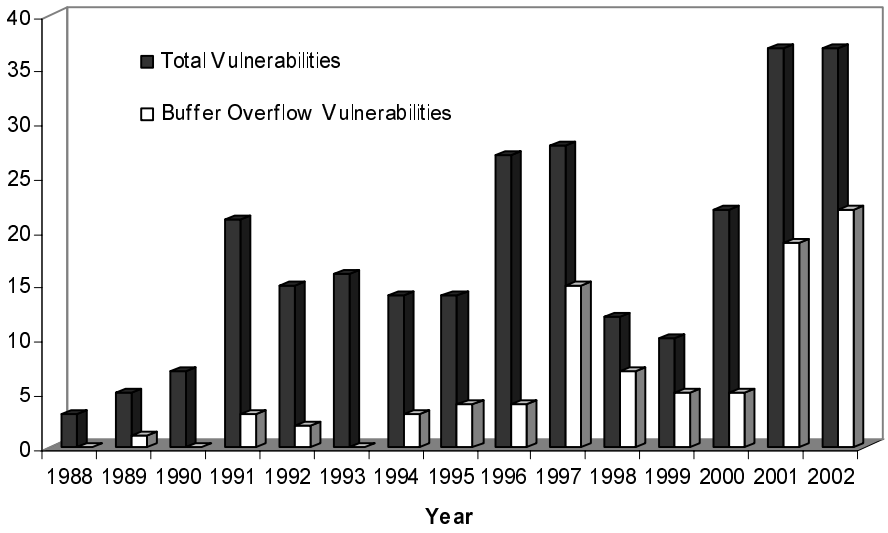

Figure 1. CERT reported vulnerabilities, 1988-2002

A buffer overflow occurs when a function writes beyond the boundaries of the destination buffer and hence overwrites the contents immediately adjacent to it. Generally this causes a memory corruption or memory fault of the program. However, it can also be exploited maliciously to alter the control flow of a program in order to break the security of the system [8,18]. Buffer overflow attacks can be generally divided into two sub-categories depending on the buffer allocation over stack and heap, thus known as Stack smashing attacks and Heap smashing attacks, respectively.

Stack smashing attack can be defined as an attack that overwrites the return address stored in the stack and this can be used in changing the control flow of the program by re-directing the return address to point to a malicious code that has been injected into the program memory through some means [20,24]. Whereas, Heap smashing attack over runs the heap to change the control flow of the program such as overwriting the function pointers stored there [26].

Top

There are four attack targets that could be victimized to change the control flow [8]:

i. *The return address, allocated on the stack:* The return address can be abused by overflowing the buffer stored in stack or another way is to write on the return address through dereference pointers.
ii. *The old base pointer, allocated on the stack:* It can be abused by building a fake stack frame with a return address pointing to attack code and then overflow the buffer to overwrite the old base pointer with the address of this fake stack frame. Upon return, control will be passed to the fake stack frame which immediately returns again redirecting flow of control to the attack code.
iii. *Function pointers, allocated on the heap, in the BSS or DATA segment, or on the stack either as a local variable or as a parameter:* If the function pointer is redirected to the attack code the attack will be executed when the function pointer is used.
iv. *Longjmp buffers, allocated on the heap, in the BSS or DATA segment, or on the stack either as a local variable or as a parameter:* They contain the environment data required to resume execution from the point *setjmp()* was called. This environment data includes a base pointer and a program counter. If the program counter is redirected to attack code the attack will be executed when *longjmp()* is called.

To cater such attacks researchers have come up with two approaches namely; static and dynamic. In the static approaches programmers analyze, personally or through some tools, the source code to predict the behavior of the program when it would be brought for execution and hence locate the security vulnerabilities that can give rise to buffer overflow attacks. Whereas, in the dynamic approaches the attacks are prevented by the changes in the run time environment, via some automated library wrappers or implementing safe compilers, in order to make the programs less vulnerable.

The rest of the paper is organized as follows; Section 2 discusses the basics of stack smashing attacks. Section 3 is about some of the renowned protection mechanisms against stack smashing attacks discussed in taxonomical manner. Section 4 contains the qualitative comparisons of selected techniques. Section 5 gives some suggestions for possible enhancements to already existing approaches and finally Section 6 concludes the paper.

2. Stack Smashing Attacks

When a function is called the return address of the calling function is stored on the stack together with the stack frame pointer. The naïve approach of launching the stack smashing attack is to overflow a buffer in such a way that the return address is overwritten and it points to the malicious code, so that when the called function returns the flow of program execution transfers to the malicious code rather than returning to the calling function [5,16,18,20]. This is the way an attacker can execute some malevolent code if it is cleverly placed in memory. This malicious code is usually a binary representation of a shell code with the help of which an attacker can spawn a new shell with the root privileges. This will give him full freedom of launching malicious activities, for example, installing some security breaching application, reading confidential information, or deleting essential information etc. Figure 2 shows a sample code for launching stack overflow attack [24]. Figure 3a and Figure 3b are the pictorial view of stack before and after the attack.

```
char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];
void main()
{      char buffer[96];
       int i;

       long *long_ptr = (long *) large_string;

       for (i = 0; i < 32; i++)
             *(long_ptr + i) = (int) buffer;

       for (i = 0; i < strlen(shellcode); i++)
             large_string[i] = shellcode[i];
       strcpy(buffer,large_string);
}
```

Figure 2. stack overflow attack

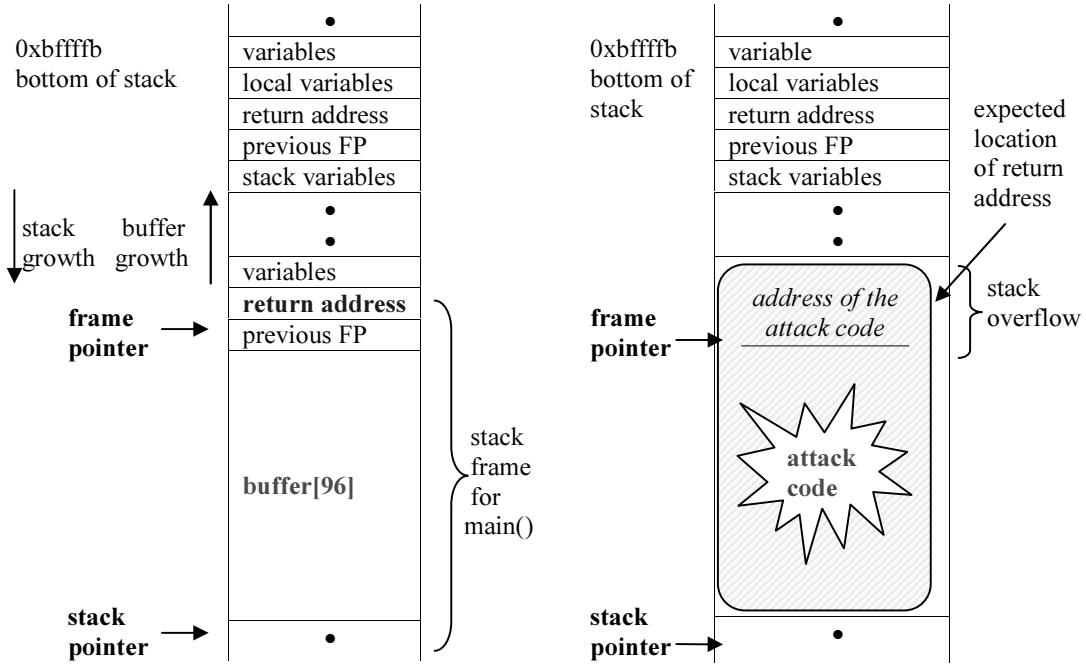

Figure 3. Stack view before and after an attack

An attacker must fulfill the following conditions in order to launch a successful stack smashing attack.

i. He must be able to find buffer overflow vulnerability in a program to prepare the platform for the attack.

ii. The size of the buffer must be determined.

iii. The attacker must be able to control the data written into the buffer. This way he will be able to insert intended malicious address at desirable code pointer (return address, frame pointers etc) locations.

Top

iv. There must be security sensitive variables or executable program instructions stored below the buffer in stack.
v. Target executable program instructions must be replaced with other executable instructions.

3. Protection Mechanism

The campaign for preventing stack smashing attacks has given rise to two fundamental schools of thought. Each of these disciplines has proposed several solutions and most of them are claimed to be promising for tackling such attacks.

Static Approaches. The stack overflow vulnerabilities can be detected and/or prevented through an analysis of the program source code. The major advantage of such approaches is that security bugs can be eliminated before application is deployed.

Besides having their own pros and cons, the static approaches discussed so far have common innate drawbacks that they require an updated database of programming flaws to test for. Also they all require complete recompilation/modification of source code for a particular application. So all the vulnerable libraries in the existing operating systems must be recompiled and deployed to replace the running copies, unfortunately this is not that easy to accomplish. Therefore, these techniques are helpful for future applications or those that are being developed. Examples include ITS4, Integer range analysis, Source to source transformation, Cyclone, CCured, Gemini etc.

Dynamic Approaches. These approaches are the runtime preventive measures for stack smashing attacks. This means that vulnerable programs become less vulnerable or harmless in dynamically protective environment which is contrary to when these programs are brought into the non-protective environment. However, these programs can only tackle the known vulnerabilities because these approaches are based upon patching up the loop holes that are exploited by the attackers. Consequently these approaches are ineffective against the unknown future attacks. Some of the examples are, StackGuard, Stack Shield, Stack Ghost, Libsafe, Libverify, ProPolice, RAD, Purify, Point Guard, Valgrind etc.

Approaches stated above have given rise to many overlapping solutions targeting the specific areas to cover overflow vulnerabilities. We can divide the solutions in three fundamental approaches, which are *Return Address Protection*, *Access Violation Probing*, and *Bounds Checking*. These approaches are discussed in detail in coming sections.

3.1 Return Address Protection Approaches

In this approach, the Return Address is protected in order to prevent the change of control flow of a vulnerable program. The control flow is changed from normal to malicious code by overwriting the return address of the calling function to that of attack code. However this approach doesn't prevent the injection of malicious code.
Canary Insertion, StackGuard. This approach is devised mainly for encountering return address overwrites attacks [1,8,11,15,18,20]. In the stack smashing attack, the attacker has to write the malevolent contents to the

dest buffer surpassing local variables, old base pointer and all the way to the return address. The core concept of encountering such attack is to put a dummy value between the old base pointer and return address. This value acts as an alarm. Upon returning from the function this value is validated, if it's tampered then a stack overflow is detected. The inventors have named this value as *canary* as shown in Figure 4.

The potential ways to circumvent this detection method is to either leave the canary word intact while changing the return address to point to a malicious code, this is made possible by overwriting the return address using an abused pointer and not touching the canary word, or by overwriting it with its correct value and thus in a way not changing it.

To overcome these shortcomings the inventors have proposed two versions of canary words, *random canary* and *terminator canary*. A *random canary* is a 32-bit random value calculated at run-time. It is very difficult for an attacker to guess this random value at run-time thus preventing it from being overwritten. On the other hand the *terminator canary* consists of string termination sequence, for example, `NULL character`, `Carriage Return`, `-1`, and `Line Feed`. Inserting the string termination symbols will prevent an attacker from reaching the return address as the library functions do not allow to go beyond string terminals. It is important to note that above solution works only in case where the attacker overwrites everything along the stack, however to cater the *abused pointer attack*, identified by [8], inventor have presented an approach that not only saves the canary value but also the *XOR* of the canary and the correct return address. This would allow the detection of change in return address while keeping the canary word intact. It is necessary for this scheme to use random sequence of canary word as the terminator canary XORed with an address would not terminate strings any more.

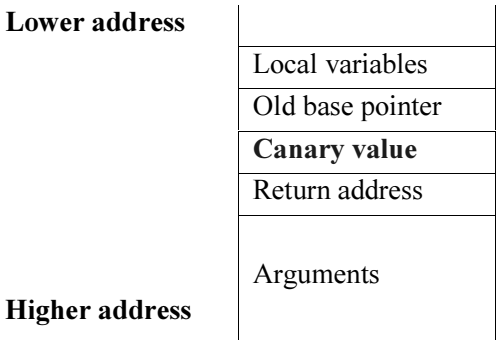

Figure 4. The StackGuard stack frame

Return Address as Canary, Libverify. The Libverify library has implemented a return address verification scheme similar to that implemented in the StackGuard. The difference is the manner of implementation. In contrast to StackGuard, which generates random numbers for use as canaries, Libverify uses the actual function return address as the canary value for each function [8,16]. Moreover, the StackGuard injects the verification code during compilation; Libverify introduces the verification code at the start of the process execution through a binary re-write of the process memory and hence does not require the recompilation of the software.

Top

To accomplish this Libverify rearranges the code a little bit. First each function is fully copied to the heap (requires executable heap) where it can be altered. Then saving and verifying of the return address is inserted into function by overwriting the first instruction of the original function with a call to *wrapper-entry* and return instruction with a call to *wrapper-exit*. The copying of functions to the heap is because of the Intel architecture unlike other platforms. Libverify does not protect the integrity of the canary stack. They propose protecting it with *mprotect()* as in the RAD but as in the RAD case this will most probably impose a very serious performance penalty. Also, it has same drawback as in the case of Libsafe.

Guard Value, Propolice. It is a GCC (Gnu Compiler Collection) extension for protecting applications from stack-smashing attacks [8,28]. ProPolice borrows the main idea from StackGuard – it places a canary value which is given the name Guard to detect attacks on the stack. The novelty is that it protects the local stack allocated variables by relocating them (irrespective of the order declared by the programmers) in such a way that char buffers come before all the other variables i.e., just after the Guard value. The re-arranged stack frame can be viewed as shown in Figure 5.

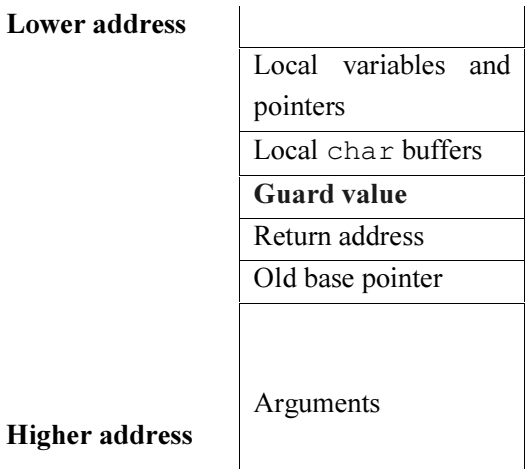

Figure 5. The ProPolice stack frame

In this way apparently it seems that local char buffers can only be over flown to perturb each other, the old base pointer and the return address, but by placing the guard value between these buffers and the old base pointer all the attacks outside the buffers segment will be detected. Once an attack is detected the process is terminated. The beauty of this approach is that unlike StackGuard, it can nicely tackle Emsi's attack [27].

ProPolice when tested in [8], it was revealed that it is somewhat unstable in the sense that it skipped the protection over small character buffers even if the user had set the protector flag.

Return Address Repository, RAD. As the name suggests this approach is implemented to protect the return address from malicious overwrites. RAD is basically a simple GCC compiler patch that automatically adds protection code into the function prologues and epilogues of the programs compiled by it [1,15]. 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. The key idea behind RAD is quite similar to return address protection of Libverify. Every time a function call is made and a new stack frame is created, RAD stores a copy of the new return address. When a function returns, the return address about to be de-referenced is first checked against its copy. There are two variants of RAD, MineZone RAD and Read-Only RAD, which protects the return addresses, stored in Return Address Repository (RAR) in different ways. MineZone RAD is more efficient while Read-Only RAD is more secure.

Global Ret Stack, StackShield. It is a GCC compiler patch for three kinds of protections [8]; two are against the overwriting of return address and one against overwriting of function pointers. Return address overwriting is protected by implementing Ret Range Check and Global Ret Stack.

This preventive approach uses a separate global array of 32-bit entries in the stack for storing the return addresses. Whenever a function call is made, the return address that is pushed onto the stack is also copied into this global space. Even if an attacker succeeds in overwriting the return address, he will not be able to get any benefit out of it as when the function returns, its address is replaced with the global stored value without any comparison with the current address.

Ret Range Check, StackShield. This approach is similar to Global Ret Stack mentioned above. The only difference is that instead of replacing the address, it checks the validity of return address and if this check is not validated then the execution is halted [8].

Return Address and Code Pointers Cryptography, PointGuard. Point Guard is a generalization of StackGuard such that it places canaries adjacent to all code pointers (function pointers and longjmp buffers) [8,18]. This way it checks the integrity of these code pointers by checking the canaries, if the canaries are intact it means there is no attack otherwise it issues an intrusion alert and exits the program. This approach is used for debugging purposes because of its execution overhead.

3.2 Access Violation Probing Approaches

These techniques deal with statically and dynamically checking the memory access limits. Among the following approaches, the first two are static and the rest are dynamic.

Type Safe Languages. Unsafe languages like C and C++ do not perform array-bounds checking, which turns out to be a security-critical issue, particularly in handling strings. The risks increase even more dramatically when user-controlled data is on the program stack (i.e., is a local variable). Most of the vulnerabilities are due to lack of safety in languages [8,14]. The safe programming languages on the other hand offer security features making them lucrative for developing extensible environments on wide variety of platforms [7,18]. Such languages provide the mechanism of isolation for untrusted code.

Code Segment Boundary Checking, StackShield. Code segment boundary checking entails storing the boundary address of the code segment some where e.g., in a processor register or in data segment. This way no

execution of instructions outside the code segment is allowed to be performed by any process, because a check is made before executing an instruction to assure that the instruction being executed is within the boundary of code segment by comparing the address of the instruction with the boundary address. Stack Shield makes it possible by declaring a global variable in the data segment, and its address is used as a boundary value [8,18]. Any reference by the function pointers beyond this value is taken as malicious activity and the process is terminated.

Run-time Memory Access Checking. Programs may incorrectly read or write memory in a variety of ways; these are called memory access errors. For example, the program may reference a block of memory which has been de-allocated through a free() call for a heap block, or because a function returned a pointer to a local variable. Access errors may result in wild pointers in the program and can cause abnormal program behavior, including wrong outputs and segmentation violations. Some kinds of memory access errors can be very hard to track down.

Comprehensive Memory Access Checking, Purify. It is the memory usage debugging tool for C programs. The basic theme for this approach is to check each and every memory access [5,11,18], not just for buffer overflows. These checks result in slower execution speed of the program. Due to this fact, this approach is not applicable in security and time sensitive applications but it is a very effective way of debugging.

Cyclone. It is a safe dialect of C [14]. It is intended to allow the secure portability of C code to it very efficiently. It defines some new pointer types and puts restrictions to them. Some tests showed that with CPU-intensive pointer operations it was over twice as slow as regular C code. Cyclone relies much over garbage collection. It contains also region-based memory management but that doesn't work well with everything.

Source Translator, CCured. It is a source-to-source translator for C [14]. It analyzes the C program to determine the smallest number of run-time checks that must be inserted in the program to prevent all memory safety violations. The resulting program is memory safe, meaning that it will stop rather than overrun a buffer or scribble over memory that it shouldn't touch. Many programs can be made memory-safe this way while losing only 10% to 60% run-time performance (the performance cost is smaller for cleaner programs, and can be improved further by holding CCured's hand on the parts of the program that it does not understand by itself). CCured finds bugs that Purify misses with an order of magnitude smaller run-time cost.

Invalid Memory Access Checking, Valgrind. This tool is used to detect invalid memory utilization at runtime without recompiling the source code [32]. It runs the code in virtual x86 CPU and detects when someone accesses un-initialized or invalid memory locations. The drawback of Valgrind is that it only checks the access to valid memory locations, it doesn't notice if one write past an array in stack or data segment if it still points to valid memory. It has some code to detect dynamically allocated, using malloc(), memory overflows better. Valgrind currently works only with Linux/x86.

Top

3.3 Bounds Checking Approaches

This technique is used to protect against the code injection attacks [1]. It provides prevention from all buffer overflow attacks, not just those that attempt to alter function activation records [22]. This kind of checking needs manual parsing by programmer in order to look for illegal buffer assignments and it is also possible to check the runtime memory usage. In the later case, it is needed to make a compiler patch. This patch works fine for all compatible programs.

Performance of this technique is proportionate to the number of array and string operations, the greater are such operations the worse the performance. Usually its cost is substantial.

Buffer Boundary Checking, Libsafe. The key idea behind Libsafe is the ability to estimate a safe upper limit for the buffers on the stack automatically at the run-time and then ensure that vulnerable function is not allowed to write anything across this and hence protecting the return address from being overwritten [8,16]. Vulnerable functions are considered to be the ones in Table 1 below.

Table 1. vulnerable C functions that LibSafe adds protection to

Function	Vulnerability
strcpy(char * dest, const char *src)	May overflow 'dest'
strcat(char dest, const char *src)	May overflow 'dest'
getwd(char *buf)	May overflow 'buf'
gets(char *s)	May overflow 's'
scanf(const char *format, …)	May overflow 'arguments'
Realpath(chr *path, char resolved_path[])	May overflow 'path'
sprintf(char *str, const char *format, ..)	May overflow 'str'

It is realized that local buffers should not be allowed to extend writing beyond the end of the current stack frame; therefore the old base pointer value is considered the boundary value. In this way a stack based buffer overflow can not over write the return address. This boundary is enforced by overloading the functions in the Table 1 while Libsafe guarantees that [16]:

i. Correct programs will execute correctly, i.e., no false positives.
ii. The frame pointers, and more importantly return addresses, can never be overwritten by an intercepted function - an overflow that would lead to overwriting the return address is always detected.

Having been through this discussion it seems that Libsafe is a promising solution to the stack smashing attacks.

GCC Extension to Automatically Perform Array Bounds and Pointer Checking. A gcc compiler patch [29] was developed by Jones and Kelly that performs complete array bounds checking for C programs. The representation of the pointers is not changed and the compiled programs are compatible with the gcc modules. They have derived a base pointer from each pointer expression, and check the attributes of that pointer to determine whether the expression is within bounds or not. The problem with this approach is its performance

cost e.g. a pointer intensive program (3D matrix multiplication) experienced 30x slow down. This penalty is proportional to pointer usage which is quite common in privileged programs. The compiler is also not mature.

Integer Range Analysis by Treating C Strings as an Abstract Type. The approach has developed a system to statically detect buffer overflow in C [17]. It can be used effectively to find both known and unknown buffer overflow vulnerabilities in a version of 'sendmail'. The approach formulates the problem as an integer range analysis problem by treating C strings as an abstract type accessed through library functions and modeling pointers as integer ranges for allocated size and length. A consequence of modeling strings as an abstract data type is that buffer overflows involving non-character buffer cannot be detected.

Source to Source Transformations. It is a system that detects unsafe string operations in C programs [13]. This system performs a source-to-source transformation that instruments a program with additional variables that describe string attributes and contains assert statements that check for unsafe string operations. The instrumented program is then analyzed statically using integer analysis to determine possible assertion failures. This approach can handle many complex properties such as overlapping pointers. However, in the worst case the number of variables in the instrumented program is quadratic in terms of number of variables in the original program. To date, it has only been used on small example programs.

Lexical Analysis, ITS4. It is a lexical analysis tool that searches for security problem using a database of potentially dangerous constructs [10]. Lexical analysis techniques are fast and simple but their power is very limited since they do not take into account the syntax or semantics of the program.

3.4 Miscellaneous Approaches

The stack overflow protection schemes presented above are not the only ones. Other researchers have come up with some different ideas which are briefly described in the following discussion.

Randomization Techniques. These techniques rely on the randomization of memory layout and the application of cryptographic techniques using some randomly generated secrets.

Memory Address Obfuscation. This technique involves the randomization of memory layout, which actually does not make attacks impossible but it makes them highly improbable. This memory obfuscation vastly increases the security. The randomization can be done for [6];

i. *The base address of the memory regions:* this involves the randomization of stack and heap base address, the starting address of dynamically linked libraries, and the locations of functions and static data structures in the executable.
ii. *Permuting the order of variables and functions:* this attempts to counter the attacks that exploit relative distances between objects. Permuting the variables allocation makes it difficult to predict the distance accurately enough to alter security-critical data without disturbing other data that may be essential for the normal flow of the program.

iii. *The introduction of random length gaps:* this incorporates padding in stack frames, padding between malloc allocations, padding between variables and static data structures, and random length gaps in the code segment.

The advantage of this approach is that almost all failed attempts result in crashing the vulnerable program and thus alerting the system administrator. A system protected with this mechanism is guarded against stack smashing attacks. It makes it almost impossible for an attacker to over write the return address because of return address randomization. However, one of the greater disadvantages of memory obfuscation is making a small portion of the virtual address space unusable, thus reducing the amount of memory available for a process.

Randomization of Executable Code. The main idea here is randomization of the code that is executed in a process [3,4]. This mechanism is very effective for code injection attacks. It is to be noted that the injected code's machine code must match with that of attacked system. The core of this scheme is to randomize the code in executable by encrypting it. This encryption is taking the XOR of executable code with a random 32-bit key. Similarly when the instruction is fetched, it is decrypted by performing XOR with that 32-bit key to get the original code. To launch a successful attack, an attacker needs to know the secret random 32-bit key, which is very difficult to break. Thus it prevents the attacker from malicious code injections.

There are two ways of encrypting the executable code, either before its execution [3] or when the file is loaded into main memory by the operating system upon execution [4]. A major weakness of the former approach is that it stores the key in the header of the executable file. Therefore if an attacker is able to get this file, he will be able to inject the needed code just as normal injection since he will gain the key for encryption and decryption operations. The later mechanism is a nice solution to this problem. In this case, it not only avoids the storage of key in executable file but also this key is different for each executable. A common drawback with both approaches is in terms of compatibility. Both needs the decryption process be accomplished by the hardware and this means that hardware vendors and operating systems vendors would both have to accept such a scheme before it could be widely deployed.

Non Executable User Stack. One of the methods to mitigate the buffer overflow attack is to make the stack non-executable. This type of defense method allows both code injection and change of Return Address thus allowing the change in control flow of the program to malicious code but it does not allow its execution completely [1], yet any attempt to transfer control to executable code placed in stack will result in system crash. Kernel patches are available for both Linux and Solaris [30,31] that make the stack segment of the programs address space non-executable. These approaches offer zero performance penalty and that programs are saved from re-compilation.

The problem with this approach is that it is not compliant with Linux Signal Handler returns. Furthermore, 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 cope up with such situations one solution is to make the stack executable on the temporary basis but this also gives the attacker a window to launch buffer

Top

overflow attack. Moreover this approach requires the operating system to be patched, which could be quite complicated due to aforementioned reasons.

Conversion of Stack Allocated Arrays into Pointers to Arrays, Gemini. This tool transforms the stack allocated arrays of C to heap in order to prevent the return address from corruption as heap does not contain return addresses [2]. The real beauty of this approach is that it prevents all stack overflow exploits whereas the drawback is that it does not prevent the heap overflow. Moreover, it prohibits the attacker from injecting executable instructions such as shell instructions.

4. Qualitative Analysis

A qualitative comparison of different techniques based on various parameters is presented in Table 2. The columns depict solutions based on taxonomy while parameters on the basis of which these solutions are evaluated are arranged in rows. These solutions employ both the static and dynamic approaches.

The code pointer protection and access violation probing involve the protection of return address, function pointers, base pointers and de-allocated memory locations. These schemes are usually platform dependent. All the static approaches here entail both the increase and the change in source code and due to these they can not be applied on the legacy systems. On the other hand, dynamic approaches do not need change in source code but may require recompilation; therefore it would be difficult to use them where source code for the legacy systems is not available. Based on the performance code pointer protection schemes can be used either as debugger or as normal security application whereas access violation probing schemes can be employed for debugging purposes only. Among all the techniques in these two categories only Cyclone and CCured are secured against all the attack targets.

The schemes addressing bounds checking are mostly static and usually platform independent. Although majority of these approaches have high performance penalty but Libsafe which is a dynamic solution offers very low performance degradation. Also Libsafe is secured against all the attack targets. Unlike static approaches in this category Libsafe is applicable to legacy systems upon recompilation.

Concluding the above analysis, it is found that Libsafe offers the least performance penalty than any other scheme. Therefore it can be used both for the development of security sensitive applications and legacy systems.

Top

Table 2. Qualitative analysis of protection schemes with respect to different parameters

Parameters \ Solutions		Code Pointers Protection	Access Violation Probing	Bound Checking	Miscellaneous		
					Randomization Technique	Non-executable User Stack	Gemini
Require Changes in	Source Code	static approaches require changes; most of the dynamic approaches do not	static approaches require changes; dynamic approaches do not	static approaches require changes; dynamic approaches do not	No	No	Yes
	Compiler	Yes	static approaches require; dynamic approaches do not	No, except Libsafe	No	No	Yes
	Operating System	No	No	No	Yes	requires kernel patches	No
	Microprocessor	No	No, except Valgrind	No	No	No	No
The Need for Source Code Recompilation		static approaches need; most of the dynamic approaches do not need	static approaches need; most of the dynamic approaches do not need	Yes	No	No	Yes
Platform Independence		usually no	usually no	Yes	Yes	No	Yes
Code Size		increases in static approaches; remains same for dynamic approaches	increases in static approaches; remains same in dynamic approaches	increases	remains same	remains same	increases
Impact on Performance		usually moderate degradation	high degradation	moderate degradation	moderate degradation	low Degradation	low Degradation
Applicability on Legacy Systems		static approaches are not applicable; dynamic approaches are applicable	static approaches are not applicable; dynamic approaches are applicable	static approaches are not applicable; dynamic approaches are applicable	applicable	applicable for dynamic approaches	not applicable
Practical Applications		depending upon performance cost, usage varies from security application to debugging tool	it is used as debugging tool	it can be used as debugging tool as well as security application	it can be used as debugging tool as well as security application	it can be used as debugging tool	it is used as security application

5. Proposed Solution

As we have seen there have been tremendous amount of efforts invested by a great number of researchers for the last 15 years to overcome buffer overflow vulnerabilities with special interest to prevent stack smashing attacks. In this struggle, several solutions have been noticed by the security community. Different researchers have invested their efforts to come up with unique solutions but there are so many solutions that are based on a single approach with slight modifications to the basic theme e.g., using canary to protect return address. In the following we propose some useful modifications to the existing solutions that would be helpful in boosting the security against the stack smashing attacks.

In Code Segment Boundary Checking to protect function pointers by Stack Shield, instead of storing the code segment boundary some where in the data segment one can slightly modify the PCB (Process Control Block) to hold the code segment boundary in the Process Identification Information block. This is because code segment boundary for a particular process never changes that's why a process can hold this information upon its creation. Unlike Stack Shield where an attacker can overflow some pointer to overwrite the code segment boundary, this will provide a better solution because it is not vulnerable to malicious over write. This solution will require a slight change in the PCB data structure and the corresponding operating system routines.

The XORing of canary with the return address and then storing the return address, canary and the XORed values by StackGuard will limit the number of times one can recursively call a particular function. Because in their case a canary is generated at each function call and is stored some where in the data segment which has limited size. It means there must be some limit for subsequently storing these canaries. In the case of excessive recursive or subsequent calls to functions a time will reach when this storage will start overflowing which means there must be some mechanism to handle this situation to avoid abnormal behavior. In our opinion the simplest solution is similar to what we have mentioned above i.e., assign the canary to process and not to each and every function. This canary can be stored in PCB and will not limit the number of recursive or subsequent calls; see Figure 6. Hence, whenever a function is called by the process we can XOR the return address with the canary and store the return address and the XORed value in the stack. This solution entails a slight change in compiler, PCB structure and the apposite operating system routines.

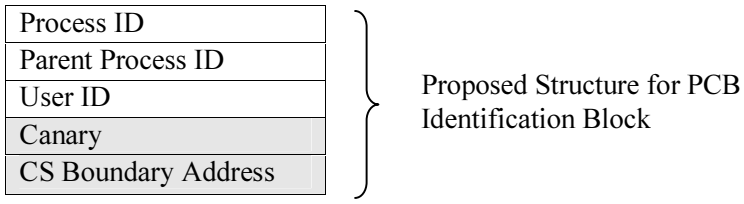

Figure 6. modified process control block

Another interesting solution would be to put a memory mirror that contains a bit, initially reset, next to each memory location as depicted in Figure 7. This will allow the operating system to deny the access to those memory locations which contain return address and old base pointer. This mechanism is very simple, while

preserving the return address and old base pointer to memory locations, upon the invocation of a function call, operating system will also set the pertinent memory bits. This will deny any malicious access to overwrite these memory locations. Upon returning from the function, the operating system will reset these memory bits to release them for later use. This technique will provide better protection than MemGuard[21], because there is no limit for recursive or subsequent function calls.

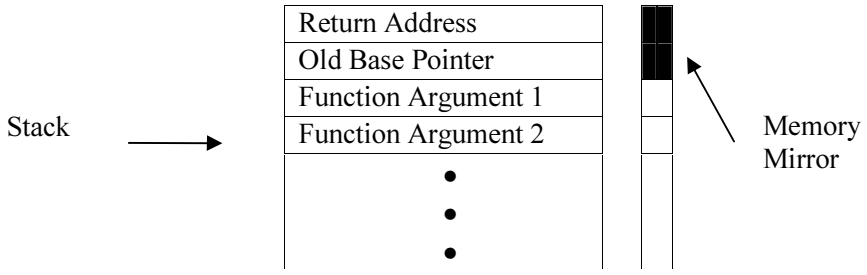

Figure 7. proposed memory architecture

6. Conclusion

One of the most common ways of breaching systems securities is to exploit buffer overflow vulnerabilities. Attackers have excessively exploited these vulnerabilities and caused various threats. According to a study carried out at University of California at Berkley, buffer overflows alone stand for about 60% of the vulnerabilities reported by CERT in year 2002. Realizing the importance of severe damages caused by this security threat, researchers have done a great deal of study in order to prevent or detect it. We have studied and analyzed some of the well know approaches. The buffer overflow attacks can be generally classified into two types as stack and heap smashing attacks. In this study we have focused on stack smashing attacks and further we have categorized them in two basic classes, namely, *Static* and *Dynamic*. Moreover we have classified them according to protection mechanism. Also a detailed qualitative comparison of different techniques is presented. This will help security researchers and designers to choose among the techniques based upon the desired parameters and system specific criteria. We have also proposed some recommendations that, if incorporated, will strengthen the security and performance of some of the existing solutions. The performance analysis of proposed solutions in comparison to approaches given in the literature could be the future work.

References

[1] Mark Shaneck, “An Overview of Buffer Overflow Vulnerabilities and Internet Worm”, *CSCI 8980*, December 10, 2003.

[2] Christopher Dahn and Spiros Mancoridis, “Using Program Transformation to Secure C Programs Against Buffer Overflows”, *10th IEEE Working Conference on Reverse Engineering, Victoria, B.C., Canada,* November 13 - 17, 2003.

[3] G. Kc, A. Keromytis, V. Prevelakis, “Countering Code-Injection Attacks with Instruction-Set Randomization,” *In Proceedings of the 10th ACM Conference on Computer and Communication Security,* October 2003.

Top

[4] E. Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefanovic, D. Zovi, "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," *In Proceedings of the 10th ACM Conference on Computer and Communication Security,* October 2003.

[5] Suan Yong and Susan Horwitz, "Protecting C Programs from Attacks via Invalid Pointer Dereferences", *The ESEC/FSE 2003 technical program on Safety and Security,* September 5, 2003.

[6] S. Bhatkar, D. Du Warney, R. Sekar, "Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits", *In Proceedings of the 12th USENIX Security Symposium,* August 2003, pp 105-120.

[7] Andrew Wright, "On Sapphire and Type-Safe Languages", *Proceedings, Texas Workshop on Security of Information Systems, Texas A&M University,* April 2 2003.

[8] John Wilander and Mariam Kamkar, "A Comparison of Publicly Available Tools For Dynamic Buffer Overflow Prevention", *The 10th Annual Network and Distributed System Security Symposium, San Diego, California,* 6-7 February 2003.

[9] B. Chess, "Improving Computer Security using Extended Static Checking", *In Proceedings of the 2002 IEEE Symposium on Security and Privacy,* May 2002

[10] John Vigea, J. T. Bloch, Tada Woshi Kohno and Gary McGraw, "ITS4: Static Vulnerability Scanner for C/C++ Code", *Annual Computer Security Applications Conference,* December 2001.

[11] Christof Fetzer and Zhen Xiao, "Detecting Heap Smashing Attacks Through Fault Containment Wrappers", *In Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems,* October 2001.

[12] Timothy K. Tsai, and Navjot Singh, "Libsafe: Protecting Critical Elements of Stacks", White Paper http://www.research.avayalabs.com/project/libsafe, August 29, 2001.

[13] N. Dor, M. Rodeh, and S. Sagiv, "Cleanness Checking of String Manipulations in C programs via Integer Analysis", *In 8th International Symposium on Static Analysis (SAS),* July 2001, pp 194 –212.

[14] D. Wagner and D. Dean, "Intrusion Detection via Static Analysis", *In Proceedings of the IEEE Symposium on Security and Privacy,* May 2001, pp 156-169.

[15] T. cker Chiueh and F.H. Hsu, "RAD: A Compile Time Solution to Buffer Overflow Attacks", *In proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS), Phoenix, Arizona, USA,* April 2001.

[16] Arash Bratloo, Navjot Singh, and Timoyhy Tsai, "Transparent Runtime Defense Againt Stack Smashing Attacks", *In proceedings of 2000 USENIX Annual Technical Conference, San Diego, California, USA,* June 18-23, 2000.

[17] D. Wagner, J. Foster, E. Brewer, and A. Aiken, "A First Step towards Automated Detection of Buffer Overrun Vulnerabilities", *In Network and Distributed System Security Symposium, San Diego, CA,* February 2000.

[18] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade", *In Proceedings of the DARPA Information Survivability Conference and Expo (DISCEX), Hilton Head, South Carolina,* January 2000, pp 119–129.

Top

[19] A. Simon, A. King, “Analyzing String Buffers in C”, *In International Conference on Algebraic Methodology and Software Technology,* 2000.

[20] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen, “Protecting Systems from Stack Smashing Attacks with StackGuard", *Presented at the Linux Expo* (http://www.linuxexpo.org/~crispin/), Raleigh, NC, May 18-22, 1999.

[21] C. Cowan, C. Pu, D. Maier, H. Hinton, J.Walpole, P. Bakke, S. Beattie, A. Grier, P.Wagle, and Q. Zhang. “Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks”, *In Proceedings of the 7th USENIX Security Symposium,* January 1998.

[22] Jones, R. W. M. and Kelly, P. H. J. "Backwards-compatible bounds checking for arrays and pointers in C programs", *In Third International Workshop on Automated Debugging, Linkoping, Sweden,* May 1997.

[23] Georgr C. Necula, “Proof-Carrying Code”, *Annual Symposium on Principles of Programming Languages, Proceedings of the 24th ACM SIGPLAN-SIGACT, Paris, France,* 1997, pp 106 – 119.

[24] “Aleph One, Smashing the Stack for Fun and Profit”, *Phrack Magazine Volume 7, Issue 49,* November 1996.

[25] *CERT Advisories*, http://www.cert.org/advisories, Feb 2003.

[26] P. –A. Fayolle and V. Glaume, “A Buffer Overflow Study, Attacks & Defences”, http://www.enseirb.fr/~glaume/indexen.html, March 2002.

[27] http://www.securiteam.com/securitynews/5TP0N0A2AW.html, Aug 2000.

[28] H. Etoh, “GCC Extension for Protecting Applications from Stack-Smashing Attacks”, http://www.trl.ibm.com/projects.security/ssp, June 2000.

[29] Richard Jones and Paul Kell, “Bounds Checking for C”, http://www-ala.doc.ic.ac.uk/phjk/BoundsChecking.html, July 1995.

[30] “Solar Designer, Non-Executable User Stack”, http://www.openwall.com/linux

[31] Casper Dik, “Non-Executable Stack for Solaris”, Posting to comp.security.unix, http://x10.dejanews.com/getdoc

[32] J.Seward and N. Nethercote, Valgrind, “An Open Source Memory Debugger for X86-Linux”, http://developer.kde.org/~swardj/

[33] US-CERT Vulnerabilities Database, http://www.kb.cert.org/vuls/

Top