The Cost of Preventing a Buffer Overflow

Dr. Anatoliy S. Gordonov

Abstract - **In the paper we have considered the main methods of buffer overflows, mitigation strategies, and their influence on the memory consumption. The analysis of various methods of stack protection has given us an estimate of the additional memory required for the implementation of specific techniques. The size of the additional memory depends on many factors including computer architecture, OS environment, programming languages used to create the program. For the protection methods considered in the paper, the cost may vary from the insignificant amount for prevention purposes, based on the careful analysis of input data in the program, to the use of Guard Pages when extra memory may include additional pages of the memory. In many cases developers have to use various mitigation strategies in order to make programs less vulnerable to buffer overflows. The main contribution of this paper is the analysis and evaluation of the additional memory required for the various methods of protection from buffer overflow. The current paper allows readers to understand the cost of these methods more clearly, which, in turn, will result in more efficient and secure programs. The results of this paper are useful for both software developers and the instructors who teach methods of secure programming.**

Index Terms- **buffer overflows, mitigation strategies, memory consumption**

I. INTRODUCTION

Buffer overflow attacks are the most common types of intrusion attacks today. A buffer overflow is a software bug that allows data to be copied to the locations in the memory, which are positioned beyond the boundaries of the original buffer, corrupting adjacent data or instructions. Not every buffer overflow leads to program vulnerability. However, it can be used by malicious users to get control over the computer system. The possibility to exploit the buffer overflow in the specific program is not an easy task and depends on many factors. These factors include computer architecture, OS environment, programming languages used to create the program, and, of course, the qualification of the attacker. One of the widely used buffer overflow variants is a stack overflow, which can be exploited in computers that support stacks. Some runtime environments for languages, such as Java, Pascal, and C#, detect buffer overflows and generate an exception. However, very popular programming languages C and C++ that are widely used in programming development do not have this kind of detection. This allows the attackers to exploit stack overflows to get control and change the execution of the program.

II. THE PROBLEM

As it was mentioned above, buffer overflows may be used in different flavors. Some of the exploits of program vulnerabilities are more complex than just a buffer overflow, but still based on it. Several types of buffer overflows are known. They are also called buffer overflow generations [1].

The first generation of buffer overflows (stack smashing) is described in [2]. This attack is based on the fact that a buffer in a procedure is allocated in the stack where the return address is saved. Writing to the buffer more items than it can hold will overwrite adjacent to the buffer areas in the stack including the return address. If an attacker were able to submit an executable code to the buffer and change the return address to the address of the beginning of this code, the code would be automatically executed as soon as the function executes the return command.

One of the variations of the traditional stack overflow attack can be used in the Windows environment, and it is based on exploiting Windows structured exception handling (SEH) [5]. This mechanism allows applications to register a handle to act on errors. Then, if an error has occurred, an application has a chance to catch an exception and recover. All handlers are connected together into the chain, and this chain is traversed to find an exception with the required code. The chain is created for each thread and is allocated in the stack area. The compiler reserves the space in the stack for local variables, which are immediately followed by the exception handler address. This gives the ground for exploit: by overflowing a local buffer, the address of exception can be changed to the code submitted by the attacker.

The second generation of buffer overflows is related to the mistake (called *off-by-one*) that can frequently be found in programs. By "*one"* we mean an element of an array. For example, very often, this error can be found in loop operators where elements may start from 0 instead of 1, or by comparing the end of the loop with "<=*n*" instead of "< *n*". Changing one element beyond the boundaries of the buffer will change the element that is located in the stack next to the end of the buffer. This can be saved "state information" that includes the saved frame pointer (EBP) and return address (EIP). In this case, EBP would be changed, and EIP would most likely stay the same. The number of bytes to be changed in EBP depends on the data type written into the buffer. The saved frame pointer value is changed. The new EBP refers to another location in the overwritten buffer where the dummy stack frame has been created. This dummy stack frame usually has a return address pointing to the shell code in the same buffer.

978-1-4799-5233-5/14/$31.00 ©2014 IEEE

This type of attack is indirect and can be used when only a limited number of the bytes in the stack can be overwritten [3].

The third generation of buffer overflows is *Arc Injection* [1, 3, and 4]. Two types of buffer overflows considered above (namely, stack smashing and *off-by-one*) are based on overwriting part of the stack with a code specially created by the attacker and passing control to this code. These type of attacks are not possible if the computer system has some form of memory protection (for example, W^X mechanism, which prevents any area from being simultaneously writable and executable). In this case, an attacker tries to find a way to run a code, existing in the memory, with specially prepared arguments. In many instances, this code is a standard library function that already exists in the memory. *Arc injection* is also called *return-into-labc* (where the *labc* is a standard UNIX or LINUX system library) or *Return to System Call.* The name *Arc injection* comes from the fact that the attack includes a new arc (transfer of control) in the program flow chart but not a new node with a code. In many operating systems, the standard C library is loaded for most processes at a well-known address and allows an attacker to find the addresses of the functions. A more advanced method of *Arc injection* allows an attacker to execute a chain of library calls, one after another.

All buffer overflow types mentioned above (as well as some others not considered here) provide a basis for many security attacks. Various mitigation strategies used to prevent and stop these attacks are known. All of them have their own price in terms of program performance. But if the evaluation of the extra time required for the execution of additional security mechanisms may be found in literature, there is almost no information about the extra memory required for this. Below, we are going to analyze the cost of different protection mechanisms from the point of view of memory usage.

III. ANALISIS OF DIFFERENT MITIGATION STRATAGIES

There are several directions of mitigation strategies: prevention, detection, and recovery. The general runtime protection strategies may be classified [6] by the various participants of the development process that provide these mechanisms: developers, compilers and associated runtime systems, the operating systems. There are also classifications of mitigation techniques [1, 7] that involve eliminating the cause of buffer overflows or dealing with alleviating the impact of buffer overflows by fixing the surrounding of a vulnerable program.

The best way of protection is to prevent buffer overflows from occurring. It can be done, for example, by carefully analyzing the length of the input in order to prevent overflows. This analysis, included in the program by a developer, introduces some additional insignificant memory overhead, which is definitely a good price for the protection from buffer overflows. Data validation requires that the developers correctly identify and check all the external inputs. Being very "simple" and obvious, this mitigation strategy is hard to be fully implemented in practice because of "the human factor" involved. Developers make errors, and this source of exploitable software flaws will not cease to exist. That does not allow us to use this strategy as the main instrument of solving the input validation problem and requires using it together with other protection mechanisms, such as manual secure code audits, automatic software tests of the source code (which are faster and more cost-effective than manual inspections), and binary audits (in cases when a source code is not known).

The use of the methods mentioned above can help find many bugs, but cannot provide a completely safe solution. One of the examples is a situation when a technique used for the attack is not known. In this case we can use techniques that eliminate the flaws by reducing the damage to a program.

Many types of stack-based buffer overflows can be eliminated by using wrapping unsafe library functions. *Libsafe* [8] offers an approach of using dynamically loadable libraries that catch and substitute calls to vulnerable library functions. Being very useful for eliminating various types of stack-based overflow attacks, unfortunately, this method does not make it possible to cover all potential unsafe library functions.

Another approach utilizes the compiler extensions with specific safety mechanisms. A well-known extension is *StackGuard* [9], which enhances the executable code produced by a compiler so that it detects buffer-overflow attacks against the stack. To do so, *StackGuard* has to detect that the return address has been altered before the function returns. It does this by placing a "*canary*" word (or *a stack cookie*) next to the return address on the stack. *StackGuard* as well as Stack-Smashing Protector "*ProPolice*" have been developed for UNIX/LINUX systems. The stack buffers overrun detection has also been developed for MS Visual Studio. The code used for canaries may be predefined or randomly generated. A predefined canary may include four different termination characters (CR, LF, NULL, and -1) that would guard against buffer overflows caused by an unbounded *strcpy( )* call. The main disadvantage of the predefined canary is that it is known to the attacker. To avoid this, random generated canaries are used. They are hard-to-spoof and are changed each time the program is executed. *Figure 1a* shows the ordinary function stack frame, and *figure 1b* shows the same frame with the added *canary*.

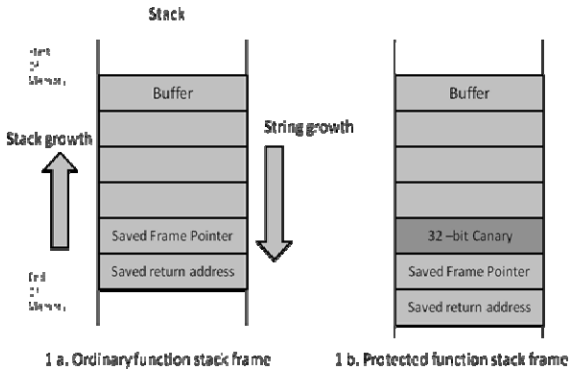

Fig. 1. Allocating a *canary* in the stack

The use of canaries includes the following operations:
Before calling a procedure:
-generate a random *canary*;
-save it in global variables;
-save it into the stack immediately after the saved return address and saved frame pointer but before the local variables in each stack frame;
Before returning from a procedure:
-compare *the canary* from the stack with the saved value;
-if they do not match, exhibit the exception code;
-exit the program if the changing of the stack area is found, or return back from the procedure.
All these steps will be implemented for every procedure call and introduce both time and memory increase. Below is an example of the program, which produces a stack overflow.

```
// compile with: /c /W1
#include <cstring>
#include <stdlib.h>
#pragma warning(disable : 4996)   // for strcpy use

// Vulnerable function
void vulnerable(const char *str) {
  char buffer[10];
  strcpy(buffer, str); // overrun buffer !!!

  // use a secure CRT function to help prevent buffer overruns
  // truncate string to fit a 10 byte buffer
  // strncpy_s(buffer, _countof(buffer), str, _TRUNCATE);
}
int main() {
  // declare buffer that is bigger than expected
  char large_buffer[] = "This string is longer than 10 characters!!";
  vulnerable(large_buffer);
}
```

Fig. 2. Example of the program with a buffer overflow.

We compiled this program with and without /GS flag and Stack Frame Check set, then, compared the sizes of the assembly codes. *Figure 3* shows an assembly listing with both /GS flag and Stack Frame Check having been set. The compiler includes extra operations that are intended to protect the stack by analyzing if the previously saved canary has been unchanged. The corresponding assembly instructions are highlighted in *figure 3*. The program compiled without setting /GS flag and Stack Frame Check does not include highlighted instructions and is 68 bytes smaller. This gives us an idea how much the stack protection costs in terms of memory consumption. It is important to note that this amount of bytes is added to every procedure call in the application. So, for instance, in our small example, this amount of memory will be doubled. In general, about 70 bytes will be added to every procedure call in an application. For big applications, the memory cost may be noteworthy.

```
16: int main() {
01391440 55                   push    ebp
01391441 8B EC                mov     ebp,esp
01391443 81 EC F8 00 00 00    sub     esp,0F8h
01391449 53                   push    ebx
0139144A 56                   push    esi
0139144B 57                   push    edi
0139144C 8D BD 08 FF FF FF    lea     edi,[ebp-0F8h]
01391452 B9 3E 00 00 00       mov     ecx,3Eh
01391457 B8 CC CC CC CC       mov     eax,0CCCCCCCCh
0139145C F3 AB                rep stos dword ptr es:[edi]
0139145E A1 00 70 39 01       mov     eax,dword ptr [___security_cookie
(1397000h)]
01391463 33 C5                xor     eax,ebp
01391465 89 45 FC             mov     dword ptr [ebp-4],eax
   17:   // declare buffer that is bigger than expected
   18:   char large_buffer[] = "This string is longer than 10 characters!!";
01391468 B9 0A 00 00 00       mov     ecx,0Ah
0139146D BE 3C 57 39 01       mov     esi,offset string "This string is ]
longer than 10 ch"... (139573Ch)
01391472 8D 7D CC             lea     edi,[ebp-34h]
01391475 F3 A5                rep movs dword ptr es:[edi],dword ptr
[esi]
01391477 66 A5                movs word ptr es:[edi],word ptr [esi]
01391479 A4                   movs    byte ptr es:[edi],byte ptr [esi]
   19:   vulnerable(large_buffer);
0139147A 8D 45 CC             lea     eax,[ebp-34h]
0139147D 50                   push    eax
0139147E E8 5E FC FF FF       call    vulnerable (13910E1h)
01391483 83 C4 04             add     esp,4
   20: }
01391486 33 C0                xor     eax,eax
01391488 52                   push    edx
01391489 8B CD                mov     ecx,ebp
0139148B 50                   push    eax
0139148C 8D 15 B8 14 39 01    lea     edx,[ (13914B8h)]
01391492 E8 E6 FB FF FF       call
@ILT+120(@_RTC_CheckStackVars@8) (139107Dh)
01391497 58                   pop     eax
01391498 5A                   pop     edx
01391499 5F                   pop     edi
0139149A 5E                   pop     esi
0139149B 5B                   pop     ebx
0139149C 8B 4D FC             mov     ecx,dword ptr [ebp-4]
0139149F 33 CD                xor     ecx,ebp
013914A1 E8 6E FB FF FF       call
@ILT+15(@__security_check_cookie@4) (1391014h)
013914A6 81 C4 F8 00 00 00    add     esp,0F8h
013914AC 3B EC                cmp     ebp,esp
013914AE E8 83 FC FF FF       call    @ILT+305(__RTC_CheckEsp)
(1391136h)
013914B3 8B E5                mov     esp,ebp
013914B5 5D                   pop     ebp
013914B6 C3                   ret
```

Fig. 3. Assembly code of the part of the program from *figure 2*.

As we can see from the above, stack canaries (or "stack cookies" as they are also called) allow the program to detect buffer overflows in the stack. In terms of additional memory consumption, using stack canaries is not very expensive. It depends on the program structure and for a program with many procedures may add a notable amount of memory. Obviously, this mechanism can work as long as the canary value remains secret. Unfortunately, using stack canaries has limitations. Neither the predefined nor a random canary offers complete protection against exploits that overwrite the return address. Canaries do not protect against exploits that modify variables adjacent to the buffer, data pointers, or function pointers. Canaries cannot prevent buffer overflows from occurring. They can only help detect buffer overflows after they have already occurred. Neither are they able to prevent

changing the return address in the situations when an attacker has managed to write directly to the location of the return address in the stack.

The goal of buffer overflows is to load an executable code (very often called *shell-code*) into the memory, overwrite the return address to point to the *shell-code*, and pass control to it. One of the major challenges that hackers meet is finding the beginning of the loaded *shell-code*. Many stack-based buffer overflow exploits rely on the fact that the stack is allocated at the known address in the memory. Being able to overwrite the function return address, an attacker can run the *shell-code*. Changing the address of the buffer in the memory by inserting randomly sized space gaps before allocating memory to the stack of a thread can make it much more difficult for the intruder to find the return address in the stack. This allocated random space is called StackGap or Guard Page. The Random Guard Page offsets the beginning of the stack, and it does not allow the attacker to predict the stack address from one run of a program to another:

-The Guard Page insertion is done by the operating system. Each Guard Page is flagged as illegal space and cannot be accessed without a process being aborted. The interesting question here is how much it costs in terms of additional memory. Each Guard Page has one memory page upper limit.

-If the Guard Page is allocated one per thread, the size of this page is the upper limit of the price that we pay. Table 1 shows the examples of various page sizes for different computer architectures.

-The standard page size is 4K (8K for Sparc). Nevertheless, as we see from table 1, much bigger pages can be used which will considerably increase the memory consumption. This may happen when threads allocate large data structures on stack and a large guard area may be needed to detect the stack overflow.

-The application that creates a large number of threads will include a Guard Page for each thread stack, which potentially results in the wasted system resources.

-Including Guard Pages between stack frames makes this mitigation strategy much stronger because it introduces more difficulties for an attacker to find absolute addresses in the stack. But, on the other hand, the price becomes too big.

Guard Pages are typically used to prevent buffer overflow attacks on global data (such as global offset table). Unfortunately, Guard Pages are not the complete solution to prevent buffer overflows. For example, they cannot prevent exploits if an attacker is able to use relative addresses.

IV. CONCLUSION

In the paper we have considered the main methods of buffer overflows, mitigations strategies, and their influence on the memory consumption. The analysis of various methods of stack protection has given us an estimate of the additional memory required for the implementation of specific techniques. The size of the additional memory use depends on many factors including computer architecture, OS environment, programming languages used to create the program. For the protection methods considered in the paper, the cost may vary from an insignificant amount for the prevention purposes, based on the careful analysis of the input data in the program, to the use of Guard Pages when extra memory may include many additional pages of the memory for each process. Not only does this memory increase the space needed for an application, but it also affects the execution time. In many cases developers have to use various mitigation strategies in order to make programs less vulnerable to buffer overflows. The main contribution of this paper is the analysis and evaluation of the additional memory required for the various methods of protection from the buffer overflow. The current paper allows us to more clearly understand the cost of these methods, which, in turn, will result in more efficient and secure programs

TABLE I
PAGE SIZE FOR DIFFERENT COMPUTER ARHITECTURES

Architecture	Page Size (KB)	Large Page Size
i386	4	4M
x86-64	4	2M, 1G
IA-64 (Itanium)	4	8K, 64K, 256K, 1M, 4M, 16M, 256M
Sparc	8	64K, 4M, 256M, 2G

REFERENCES

[1] Matthias Vallentin " On the Evolution of Buffer Overflows", 2007, http://matthias.vallentin.net/course-work/buffer_overflows.pdf

[2] Aleph One. Smashing the stack for fun and profit. *Phrack*, 49–14, November1996. Available from http://www.phrack.org/issues.html?issue=49&id=14

[3]. W. Stallings. L. Brown, "Computer Security", Prentice Hall, 2008, ISBN-13: 978-0-13-600424

[4] J. Pincus, B. Baker "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns", IEEE Security and Privacy, July/August 2004.

[5] M. Down, J. McDonald, J. Schun, "The Art of Software Assessment", Addison Wesley, 2007, ISBN 13:978-0-321-44442-4

[6] R. Seacord "Secure Coding in C and C++", Addison Wesley, 2013, ISBN-13:978-0-321-82213-0.

[7] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[8] A. Baratloo, N. Singh, and T. Tsai. Libsafe: Protecting critical elements of stacks, 1999. http://www.orkspace.net/secdocs/Unix/Protection/Description/Libsafe%20-%20Protecting%20Critical%20Elements%20of%20Stacks.pdf

[9] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78. San Antonio, Texas, Jan 1998.