Different Techniques to Prevent Buffer Overflow

Prepared By :

Kamanashis Biswas
kabi06@student.bth.se
Blekinge Tekniska Högskola
Ronneby, Soft Center
Computer Science

I. Introduction

For the past several years Buffer Overflow attacks have been one of the great critics in the filed of computing system's security. Many of these attacks have been amazingly successful, even allowing the attacker to obtain administrator privileges on the attacked systems. I have tried to review several promising methods working in practice. In this section I have described the basic idea of buffer and buffer overflow and in the next section, I have taken a look on the different methods against buffer overflow. Finally I have drawn a conclusion of my overall analysis.

1.1 Buffer

In the context of programming convention, a buffer is a contiguous allocated block of memory, such as an array or a pointer in C. More precisely, it can be said that a buffer is a region of memory which holds I/O data during execution of a program or process. That means, it is a temporary data storage area.

1.2 Buffer Overflow

A buffer overflow is a phenomenon that takes place when a program or process stores more data in a buffer than it can hold. Since buffers are designed with a finite size, usually it overwrites the memory addresses which may hold some valid data or instructions. Basically buffer overflow can be occurred through *stack overflow* or *heap overflow.* Though buffer overflow occurs due to lack of carefulness or accidentally but it can done a lot of harms such as sending new instructions to the affected systems through which he/she can corrupt user's files, delete/change valuable data or retrieve secret information. In the meanwhile, buffer overflow has raised the software vulnerabilities and has become a great panic in case of software security.

1.3 Stack Overflow and Heap Overflow

Buffer overflows are generally broken into two categories in terms of memory location. While there is no formal definition, buffer overflow can be divided into : Stack Overflow and Heap Overflow.

1.3.1 An Example of Overflowing a Stack Buffer

The following program declares a buffer that is 256 bytes long. However, the program attempts to fill it with 512 bytes of the letter 'A' (0x41).

```
int main(){
       int i ;
       char buffer [ 50 ] ;                    // create a buffer in memory
       for ( i = 0 ; i < 100 ; i++ )           // loops for 100 times
            buffer[ i ] = 'X' ;                // copy the letter X to the buffer
     return 0;
}
```

When the program executes, it copies extra data to the buffer than its limit. Eventually, the buffer overflows and it overwrites the adjacent memory area allocated by other valuable data or instruction. Even by overwriting the return IP (Instruction Pointer, sometimes known as Program Counter or PC), an attacker can change what the program should execute next. Thus he/she can run his/her program which may be destructive for the system.

1.3.2 An Example of Heap Overflows

A heap is a memory area that has been allocated dynamically. Heaps are dynamically created (e.g., new, malloc) and removed (e.g., delete, free). In some cases, it is deallocated by the programmer or garbage collector (best example in Java). Heaps are necessary as the memory-size needed by the program is not known former or it may require large memory than stack.

Heap overflow is as same as stack overflow. When a program copies data without checking whether it can store or not in the given destination, then the attacker can easily overwrite data and instruction in heap. Heap overflow is really difficult if the attacker does not have the source code or he/she needs a buffer that can overflow in such a way that it overwrites the target variables or memory address.

Here we will discuss about an example of heap overflow [2]. The program allocates memory for two buffers dynamically. One buffer is filled with 'A's. The other one is taken in from the command line. If one types too many characters on the command line an overflow will occur.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main(int argc, char **argv)
{
char *buffer = (char *) malloc(16);
char *input = (char *) malloc(16);
strcpy(buffer,"AAAAAAAAAAAAAAA");    // Use a non-bounds checked function
strcpy(input,argv[1]);
printf("%s",buffer);
}
```

With a normal amount of input, memory will appear as follows:

Address Variable Value
00300350 Input BBBBBBBBBBBBBBB
00300060 ????? ????????????????
00300370 Buffer AAAAAAAAAAAAAAA

However, if one inputs a large amount of data to overflow the heap, one can potentially overwrite the adjacent heap.

Address Variable Value
00300350 Input BBBBBBBBBBBBBBB
00300360 ?????? BBBBBBBBBBBBBBB
00300370 Buffer BBBBBBBBAAAAAAAA

1.4 Various Types of Buffer Overflow Attacks

A buffer overflow attack may be two types. One is remote and another is local. In case of remote attack, the attacker uses network port, channel to achieve unauthorized access and tries to get root/administrator privileges. It is very common today as the use of Internet spread widely in practice. On the other hand, in a local attack, he/she gain direct access of the target system, and then enhances his access privilege. Summarizing, we can say that a buffer overflow attack usually consists of three parts:

1. Putting the desired code to the target program,
2. The actual buffer overflow by copying more data in buffer that overwrite the adjacent addresses and
3. The takeover program's control to execute attack code

1.5 Details of Buffer Overflow Attacks

It is also a fact to know where the buffer is allocated. It may be a local variable of a function that resides on the run-time stack. Again it can be heap where the data and code is loaded.

In a C program, during function call the activation record or state of the function is pushed on the run-time stack. This is why the program can retrieve that sate while come back in the main function. This state can be described as follows :

1. memory allocation for each parameter,
2. the return address,
3. the base address,
4. memory allocation for local variable of the function.

Due to this mechanism, overwriting the return address, the attacker can easily get the access of the target program. Again by using *payload string* buffer overflow can take place taking the advantage of vulnerable function of C language.

Some of the open source operating system such as Linux, OpenBSD, Free BSD, and even Solaris also increase the risk of buffer overflow. Because the attacker need not to know a lot about how the operating

system works. The attacker can start with a long list of *no operation* instructions to pass the control to specially designed shell program that just resides after the no operations. This technique was followed in Morris worm. Thus, attackers always have an opportunity to find out the bugs and causing buffer overflow.

II. Different Techniques to Prevent Buffer Overflow

In this section, the emphasis is given on scientific approach which can be used to resist buffer overflow. Here only the tools that can be applied by the programmer are presented as our main target is to prevent it in software. Most approaches against buffer overflow are concentrated on the C language as a lot of security problems are caused by C programs. At fist, take a look on classification of the approaches:

2.1 Classification

The techniques can be divided into two categories. One is static and another is dynamic. In static approach check is done before or during compilation. On the other hand, in dynamic approach definitely check is done on runtime. The classification is shown in figure 2.1 [13].

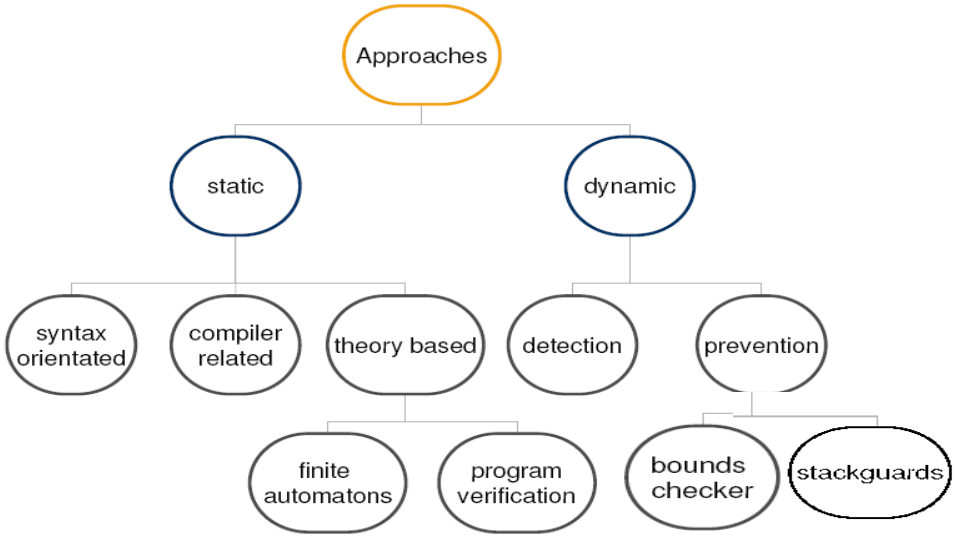

Fig 2.1 : Classification of approaches undertaken against buffer overflow

2.1.1 Static Approach : Syntactical Analysis

A lot of security problems of C program are caused by some vulnerable library functions. C library functions such as gets(), strcpy(), strcat(), sprintf(), vsprintf(), fscanf(), sscanf(), vscanf(), vsscanf(), vfscanf(), realpath(), getopt(), getpass(), streadd(), strecpy(), strtrns(), syslog() are responsible for buffer overflow in most of the time. The scanf () family of functions may also result in buffer overflows. Hence, the best way to deal with buffer overflow problems is to not allow them to occur in the first place. So take special care when using this function and it is best to use strncpy(), strncat(), snprintf(), vsnprintf() instead of above vulnerable functions. There are other functions in Microsoft's libraries that cause the same sort of problems on their platforms (these functions include wcscpy(), _tcscpy(), _mbscpy(), wcscat(), _tcscat(), _mbscat(), and CopyMemory()). Some tools for syntactic analysis are given below :

- Flawfinder
  - This software is written in Phython by David Wheeler,
  - Uses a build-in database for C/C++ functions,
  - Checks vulnerabilities for buffer overflow risks as well as format string problems, race conditions, potential shell metacharacter dangers and poor random number acquisition.
- ITS4
  - Developed by Cigital to check C/C++ codes
  - It is command line static process that checks for potentially dangerous function calls
  - It generates a report describing potential problem and providing suggestion.

- It also checks for some TOC-TOU (Time-of-Check-Time-of-Use) problems

♦ RATS
- RATS (Rough Auditing Tools for Security) also finds out potentially dangerous function calls,
- Differentiates between heap- and stack allocated buffers,
- It works for C, C++, Python, Perl and PHP

PSCAN and LCLint are also good auditing tools used for predicting buffer overflow. There are also many approaches present which work as a wrapper (also can be called middleware) of several library functions known to be vulnerable to stack smashing attacks. Some of such system based tools are SafeStr, C++ std::string, LibSafe, glib etc.

2.1.2 Static Approach : Compiler Related Approaches

Most of the earlier languages do not use bound checking. This leads to the buffer overflow. As an example we can say about Java language in which buffer overflow is impossible due to automatic bound checking. In many programs such as C uses instructions like *strcpy*(char* *dest* , char* *src*) where the two arguments are just pointers, and it is impossible for the compiler to determine the lengths of the corresponding arrays. So the compiler cannot check the range in runtime. Some tools for compiler related approaches are given below:

⇨Buffer Overrun detectiON(BOON)
- It is an automatic tool which checks only char-buffers and buffer overruns caused by library functions,
- It examines the code for potential C String overflows,
- It ignores Control flow.

⇨CQUAL (2001)
- It detects format string (functions such as fprintf, printf, sprintf, snprintf, vfprintf, vprintf, vsprintf, vsnprintf, setproctitle, syslog, and others) vulnerabilities,
- Specifies and checks the properties of C program,
- It also requires user defined type qualifiers to check the annotations

2.1.3 Theory Based Approaches

Theory based approaches are based on the concepts of theoretical computer science. It can be classified into two categories.

i) Finite Automation :

a) XGCC – uses finite automation and exploits language perspective to track assurance of conditions. In this approach, a control flow graph is generated from the source code and the tool travels through the graph using depth first search (DFS). If a branch in the graph is encountered all automation which are affected by the branches, body are duplicated. Actually it maps behavioural properties (no allocated memory should be freed more than once) to pure language properties (*free* is not applied more than once to any memory allocation point on any path through the program).

b) MOPS – MOdel checking Programs for Security properties uses a comparable approach to XGCC. It is designed to check the violation of rules. For every checked property an FSA (finite state automaton) is constructed and the program is compiled into a PDA (push down automaton).

ii) Program Verification :

a) Splint
- Static auditing tools used for checking vulnerabilities,
- Coding mistakes also informed by this tool,
- Searches for probable errors but it requires additional annotations.

b) Eau Claire
- It is a security tools that developed for finding errors in C programming language,
- It automatically checks for array bounds errors and null pointer dereferences,
- In this approach the function calls of a program are translated into the functions pre- and post-conditions.

- It works in two phases, C code to Guarded Command (a translation of an instruction into its pre and post condition) and Guarded Commands to verification condition.

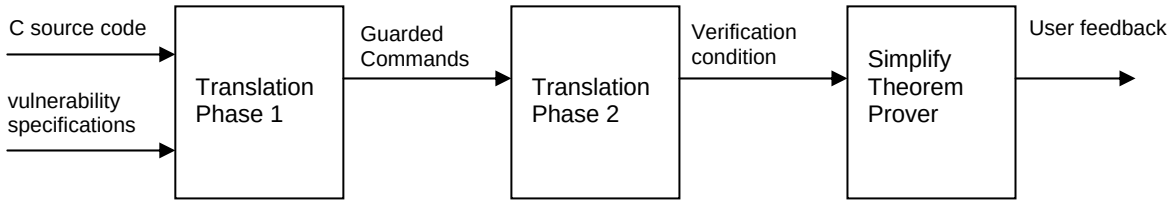

Fig : 2.2 Pictorial view of Eau Clarie phase analysis [13]

2.1.4 Dynamic Approach : Dynamic Detection

Dynamic approaches work on runtime. During program execution buffer overflows are detected or prevented here. One of the examples of this approach is Systematic Testing Of Buffer Overflows (STOBO).

⇨STOBO : It is dynamic extension of BOON's approach developed by Haugh and Bishop which aids in detection of buffer overflow problem. It keeps track of [memory] buffer lengths. The tool is supposed to accompany program testing. In this method,

- Before compilation, the source code is altered,
- It keeps records of all buffer sizes in a global table,
- Susceptible functions that may lead to buffer overflows are wrapped,
- If a wrapped function detects a potential overflow a warning message is provided
- It only detects the vulnerabilities produced by C library functions

Examples of STOBO analysis [13]:

Source code
```
char buf[100];
char *ptr;
ptr = malloc(20);
strcpy(ptr,buf);
```

STOBO output
```
char buf[100];
__STOBO_stack_buf(buf, sizeof(buf));
char *ptr;
ptr =
__STOBO_const_mem_malloc(20);
__STOBO_strcpy(ptr,buf);
```
STOBO is able to notice the possible buffer overflow

2.1.5 Dynamic Approach : Dynamic Prevention

In this approach, activities are taken earlier to prevent unavoidable situations such as program or system crash. The staple concentration is given on stack because it is the most susceptible area that can be overflow. The methods are as follows :

⇨ StackGuard and Stack Protector : Cowan, Wagle, Pu, Beattie and Walpole [4] devised a fresh approach to overcome the hijacking return address problem from stack. In this approach, the return address on the stack is protected from being altered by placing a *canary* word just next to the return address on the stack. Two types of canary is used : Terminator canary (\0, CR, LF, -1 (EOF)) and Random canary (a 32 bit random number is chosen during runtime); Null canary is not used nowadays. Here a pictorial view is presented depicting the real scenario in fig: 2.3 and 2.4. Except Canary method, there are also many stack protectors methods such as ProPoliceSSP (by IBM), Stack Shield and so on.

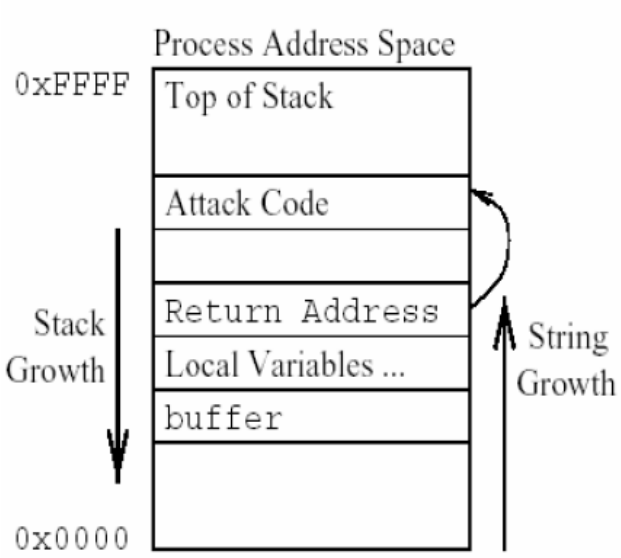

Fig 2.3 : Stack Smashing Buffer Overflow Attack

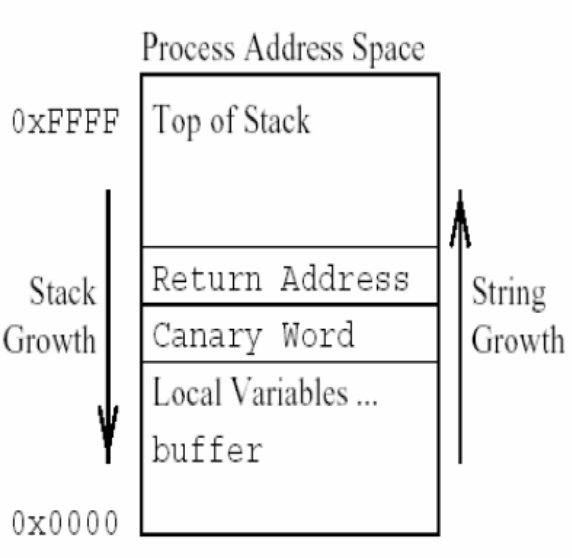

Fig 2.4: Canary word next to return address

⇨ Return Address Defender : It contains a global inter array called Return Address Repository (RAR) which consists all return addresses pushed on stack. It protects from copying of the return address and also long jump pointer based attacks. It can only work against modification of return address but cannot prevent buffer overflow attack. There is two version of RAD, MineZone RAD and Read-Only RAD.

⇨ Memory Access checking : It is a real time access violation checkers that includes verification code to the binary during program compilation. One example of this tool is 'Purify'.

2.1.6 Dynamic Approach : Bounds Checker

⇨ CRED (C RangE Detector) : All manual and automatic attempts to prevent buffer overflow is not enough to trace the vulnerabilities. Still, vulnerability is found that is exploited by the attackers. The CRED technology is a dynamic bounds checker which does not break existing code as other dynamic approaches. It keeps the track of base pointers. If a pointer is referenced then it checks whether it is valid or not. Again when any pointer dereferenced, it immediately replaces free () as it does in case of malloc (). CRED method is superior to other dynamic tools in auditing software as the overhead is greatly reduced without sacrificing protection policies.

2.1.7 Combined Static and Dynamic Approach

⇨ CCured : It is a runtime checking tool that focuses on memory safety guarantees to C programs. In this approach, pointers are classified into three categories and separated according to their usage; SEQ : Sequence pointers – pointer may be subject to pointer arithmetic but not to typecast, SAFE : Pointer remains unaltered and subject to dereference but not to pointer arithmetic or type cast, WILD: Pointer may be subject to type casting but expensive to use.

2.1.8 Commercial Tools

Besides these there are also many commercial tools that can be used to keep the code safe. These tools are most often module based and support multiple languages such as C, C++, Java, JSP, PL/SQL, C#, XML etc. Some of them not only check Buffer Overflows but also Privilege Escalations, Race Conditions, Improper Database Access, Insecure Cryptography, XSS, Insecure Account/Session Management, Command Injection, Insecure Access Control, DOS, Error Handling Problems, Insecure Network Communication, Poor Logging Practices, SQL Injection, Native Code Vulnerabilities, Dynamic Code Vulnerabilities and so on. Examples of commercial tools are Fortify, Prexis, Coverity, CodeAssure, AppDefence Developer, SPI Dynamics etc.

III. Conclusion

From 1988 to 1996 the number of buffer overflow problems remains relatively low but after 1996 it rapidly grows and becomes a crucial security issue. The technical knowledge increases as well as the development of many programming language. The attacker also finds out various ways to exploit the security problem in application programs written in these languages. Meanwhile other groups of people tried their best to face the challenge. As a result many techniques also invented to overcome this problem especially buffer overflow. In the above discussion, I have tried to give an overview of these methods. But it is not very specific as it requires a broad description to explain each tool perfectly. Moreover there is a lot of tools in practice to prevent buffer overflow. This is also a limitation of my work that I could not clearly explain all of the terms clearly. However, the buffer overflow can be avoided by writing secure code (may use common auditing tools) and the whole responsibility is of the programmer. If the programmer takes proper steps and care while coding then buffer overflow can be eliminated. But it is really tough in practice. Just observe the following code which I directly placed here without any alteration [2].

Off-By-One Overflows

Errors in counting the size of the buffer can occur usually resulting in a single byte overflow known as an off-by-one. Consider the following program where the programmer has mistakenly utilized ‘less than or equal to’ rather than simply ‘less than’.

```
#include <stdio.h>
int i;
void vuln(char *foobar)
{
char buffer[512];
for (i=0;i<=512;i++)
buffer[i]=foobar[i];
}
void main(int argc, char *argv[])
{
if (argc==2)
vuln(argv[1]);
}
```

So, minor error can cause a great disaster. That’s why; buffer overflow should be treated specially and with special care.

IV. Bibiolography

1. [Istvan - January 31, 2001] Istvan Simon, "*A Comparative Analysis of Methods of Defense against Buffer Overflow Attacks*,", California State University, Hayward, web page: http://www.mcs.csuhayward.edu/~simon/security/boflo.html, Last checked : March 24, 2006.

2. [Eric Chien and Péter Ször - September 2002] Eric Chien and Péter Ször, *"Blended attacks exploits, vulnerabilities And buffer-overflow techniques in Computer viruses",* Virus Bulletin Conference, Symantec Corporation 2500 Broadway, Suite 200 Santa Monica, CA 90404, USA.

3. [Sandeep Grover-2003-03-10] Sandeep Grover, *"Buffer Overflow Attacks and Their Countermeasures"*, http://www.linuxjournal.com/article/6701, Last Checked : March 24, 2006.

4. [Cowan, Wagel, Pu, Beattie, Walpole - 1999], Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole, "*Buffer Overflows:Attacks and Defenses for the Vulnerability of the Decade*", DARPA Information Survivability Conference and Expo, Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology,

5. [Liang, Sekar]*Automatic Generation of Buffer Overflow Attack Signatures: An Approach Based on Program Behavior Models,* by Zhenkai Liang and R. Sekar, Department of Computer Science, Stony Brook University, Stony Brook, NY 11794

6. [Goichi, Kyoko and Ichiro - 2005] NAKAMURA Goichi, MAKINO Kyoko, and MURASE Ichiro, *"2-4 Buffer-Overflow Detection in C Program, by Static Detection",* Journal of the National Institute of Information and Communications Technology Vol.52 Nos.1/2 2005

7. [Howard and LeBlanc] Michael Howard and David LeBlanc, *Writing Secure Code,* 2nd edition, Microsoft Press, 2003.

8. [Viega and McGraw] John Viega, and Gary McGraw, *Building Secure Software*, 2nd edition, Addison-Wesley, 2002.

9. The free Encyclopaedia, http://www.wikipedia.org, Last checked : March 24, 2006.

10. [Wheeler] David A. Wheleer, "Flawfinder", http://www.dwheeler.com/flawfinder/, Last Chesked : April 2, 2006.

11. http://sctest.cse.ucsc.edu/chess/EauClaire/ , Last Checked : April 2, 2006

12. [Wheleer] David A. Wheller, "*Secure Programming for Linux and Unix HOWTO",* http://docsrv.caldera.com:8457/en/SecureProg/tools.html. Last Checked : April 2, 2006.

13. [Johns - 2005] Martin Johns, *"Finding and Preventing Buffer Overflows"*, Lecture Notes, University of Hamburg, UH, FB Inf, SVS, 2005