Buffer Overflow Vulnerability Detection in the Binary Code

Shehab Gamal El-Dien, Reda Salama, Ahmed Eshak

shehab@ispofegypt.com, redasalama@hotmail.com, a_issac@sakhr.com

Al-Azhar University, Faculty of Engineering, Computers and Systems Department

Abstract

Nowadays, the Internet plays an important role in information processing and data exchange. The Internet is mainly composed of clients, servers, routers, and software applications. This internet schema is not secure due to a lot of security holes that already exist in any of the servers, the clients, the routers or the applications. One of the most dangerous security holes is Buffer overflow hole which is responsible for about 35% of the Internet attacks. A lot of solutions have emerged to detect buffer overflow security holes and protect against buffer overflow attacks such as Source Code Static Analysis, Disabling Stack Execution, Compiler-Based Techniques and Dynamic Protection solution.

The paper proposes a new solution to pre-detect the buffer overflow security holes in the binary files. So doing, the security administrator can pre-detect the buffer overflow holes in the running applications before they are exploited by the hackers. Furthermore, the solution can be used by the software developers to detect the buffer overflow security holes in their developed binaries even if they are embedding static libraries. To accurately detect the buffer overflow vulnerable applications, the proposed solution performs intensive analysis using a lot of auxiliary techniques like Call Graph, Control Flow Graph and Data Flow Graph.

Introduction

The origin of Internet attacks is the vulnerabilities and security holes that already exist in the Internet servers, clients, routers, and software applications which allow the hackers to perform their malicious activities. By exploiting the security holes, the hackers can do a lot of destructive influences on the Internet servers and clients such as denial of service and system damage attacks. In addition, the hackers can do what is more dangerous than the destructive influences, i.e. the sensitive information spoofing. Although the Internet attacks techniques are some what complex and may need a lot of efforts to be performed, the starting point for any attack is a security hole that already exists in the Internet components.

The first defense line against Internet attacks is the Firewall. The Firewall analyzes all the incoming packets to a private network and permits or drops the packet based on its own configuration rules. As an advanced detection technique against Internet attacks, the Intrusion Detection System analyzes the packets already entered to the private network (Network Intrusion Detection) or analyzes the operating system log files (Host Intrusion Detection) to detect any suspicious behavior based on his own signatures, and then notifies the administrator with the suspicious behavior.

Although all these defense techniques and devices, the Internet is still vulnerable to a lot of attacks and crimes due to the fatal security holes that already exist in the Internet components.

The Internet applications security holes represent 70% of the total holes causing the Internet attacks. One of the most dangerous applications security holes is "Buffer Overflow Hole" which is the topic of this paper. The application is buffer overflow vulnerable, if it doesn't check the size of the user input for a buffer array and the size of the input data is larger than the size of the buffer array where the areas adjacent to the buffer array will be overwritten by the extra data. The hackers exploit this vulnerability by overwriting a buffer adjacent to sensitive data such as the instruction pointer to change the program control flow.

In spite of the defense techniques (Firewalls and IDS), exploiting buffer overflow vulnerabilities can't be avoided because of the following:

- The vulnerable applications are running on the victim servers. The hacker needs only "Execute" permission on the vulnerable function to exploit it. He can get this permission if he is an authenticated user or by spoofing an IP address of an authenticated user (thus the Firewall can't block this call).
- While exploiting the hole, the hacker doesn't generate any malicious behavior and so the intrusion detection system can't detect the attack.

Several solutions have emerged to detect the buffer overflow vulnerability and prevent buffer overflow attacks. These solutions can be can be classified into, source code static analysis techniques like ITS4 [Viega 2000] and MOPS [Chen 02] and prevention techniques (Run-Time solutions) which protect the Internet servers from the attacks which occur due to buffer overflow vulnerable running application on these servers.

The prevention techniques are re-active and have a valuable performance overhead. While the source code static analysis techniques are pro-active, they have a lot of limitations.

First of all, today's applications are so complicated, perform a lot of functions and may embed several static libraries where their source code is not available. So even if the application developer applies the buffer overflow vulnerability analysis techniques on his source code, he can't guarantee that final the produced binary is free of this vulnerability because he can't apply these techniques on the static linked libraries of his binaries.

Second, even if the application vendor guarantees that the provided application is free of buffer overflow vulnerabilities, the software production process has a lot of stages and the released packages may be followed by service packs or hot fixes to fix problems. Human errors in applying the buffer overflow detection techniques may occur which means that from security point of view there is no way to guarantee that the checked source code's binary is the one we are running on the Internet servers.

Third, for a lot of reasons the applications source code may be lost (a backup is never saved or it had been destroyed, A key employee leaves without documenting a program), so the source code static analysis techniques can't be applied.

Fourth, the security administrator has no way to pre-detect the vulnerable applications, so he can't assist the running applications against this type of attack. If he doesn't apply one of the prevention solutions, the hosts may be attacked, the hosts' services may be denied or the hosts' critical information may be thieved.

Thus, although the source code static analysis techniques are pro-active, they need the source code and must be applied during the development or the testing phases.

Here the question arises, Is it visible to analyze the applications to detect the buffer overflow security holes without the availability of the source code? That is the target of our paper, analyzing the binary code to detect the buffer overflow vulnerabilities.

By applying the proposed solution, the following will be available:

- The software vendors can analyze and detect any buffer overflow vulnerability in any third party modules
- The software vendors can analyze and detect any buffer overflow vulnerability in the binary files even if they embed static libraries.
- The security administrators can detect the buffer overflow vulnerability in any binary file running on their critical servers.

The first step in the proposed solution is to convert the binary file into readable format so it can be analyzed. The binary file is disassembled, and then a simple buffer overflow vulnerability detection algorithm is applied. If the simple detection algorithm can't detect the vulnerability, an advanced detection algorithm is called. The advanced detection algorithm, either analyzes the vulnerability segment of code or it may have to analyze the whole application to detect the vulnerability. To accurately detect the vulnerable applications, a lot of auxiliary techniques are used in the advanced detection module like Call Graph, Control Flow Graph and Data Flow Graph.

The paper is divided into, in addition to this introduction, seven other sections. Section 2 explains the details of buffer overflow attacks, section 3 explains their current solutions, section 4 briefly explains the proposed solution, section 5 defines the different security rules which must be followed by any application to be free of buffer overflow vulnerability, section 6 explains the difficulties of checking these security rules, section 7 explains the details of proposed solution modules and section 8 provides an example for a vulnerable application and the rule of each module to detect the vulnerability.

2. Buffer Overflow Attacks

If a program doesn't check the size of the user input for a buffer array, and the size of the input data is larger than the size of the buffer array, then areas adjacent to the array will be overwritten by the extra data. The lack of such bound checks creates the breeding ground for buffer overflow attacks.

If the overflowed buffer is a variable allocated on the program's run-time stack such as local variables or function arguments, the attack is called Stack Overflow or Stack Smashing. As opposite to stack overflow if the overflowed buffer allocated in the Heap, the attack is called Heap Overflow. Heap overflows are generally much harder to exploit than stack overflows. The scope of this paper is the stack overflow where it is the most common and easy to be exploited.

2.1 Why does Stack Overflow occur?

Stack Overflow occurs because of C compiler doesn't perform array bounds checking, some C functions such as strcpy and strcat don't check the length of the source buffer before copying it to the destination buffer and the C programmers don't do this check. Although the destination buffer may be copied into using other C techniques other than using the vulnerable C functions but using these functions is the most common method to copy into buffer arrays.

2.2 Dangerous of Stack Overflow from security point of view

What happens when a buffer is overflowed?

The adjacent data will be overwritten which leads to unexpected behavior for the application containing the overflowed buffer. The dangerous of buffer overflow increases when the adjacent data to the overflowed buffer is sensitive, which is obvious in the process's stack. C

and C++ compilers allocate space for local variables and the return address of a function in the same stack frame which leads to changing the value of the return address if one of the locale variables had been overwritten.
By overflowing a return address of function activation record, the hacker can do:

a- Crash the application

If the new return address is a non-valid random address, the application will crash leading to denial of service Attack.

b- Re-direct the application to run its own code

In this case, the hacker redirects the application to run his own code already in the attack buffer to do his malicious code.

c- Re-direct the application to run an application resided in the memory

In this case, the hacker redirects the application to run an application in the memory to perform his malicious behavior.

3. Related work

To avoid the buffer overflow security problems, several solutions have emerged to detect buffer overflow vulnerabilities and protect against buffer overflow attacks. A brief description of these solutions is to be presented in this section.

3.1 Source code Static analysis

Several source code static analysis tools have emerged to detect buffer overflow vulnerabilities while development and testing phases. These tools can be classified into fault injection tools [Ghosh 98] which inject deliberate buffer overflow faults at random to detect the vulnerable applications and static analysis tools which statically analyze the source code to detect the buffer overflow vulnerabilities like [Wagner 2000], ITS4 [Viega 2000] and MOPS [Chen 02].

3.2 Disabling Stack Execution

Since some forms of buffer overflow attacks rely on code to be injected into the buffer and then executed, a simple solution is to install the operating system with stack execution disabled. [Simon 01]

3.3 Dynamic protection (C Safe library)

A much more robust alternative would be if we could provide a safe version to the C library functions on which the attack relies to overwrite the return address [Simon 01]. The much more C Safe library had been introduced in LibSafe library. LibSafe is a run-time solution that inserts wrapper code at the start of functions that are deemed to be vulnerable to buffer overflows. The solution details are explained in [Tsai 02] and the advantages are explained in [Simon 01] while the disadvantages are explained in [Fayolle 02]

3.4 Compiler Techniques

[Cowan 2000] devised a fresh approach to the problem. The key idea of the technique is simple.
It is based on the assumption that if a buffer overflow attack took place then everything between the buffer and the return address is likely to be corrupted. They propose to modify the compiler so that it protects the critical return address and dynamic link part of the activation record by allocating an extra field aptly called the *canary* after the dynamic link and before the local variables in the activation record. When the activation record is pushed on the stack a value is stored in the canary field. Before the function returns the integrity of the canary is checked. If it was corrupted the canary sings and the attack is detected. [Speirs 05] lists all advantages and disadvantages of this protection.

3.5 Protection through the Operating System kernel

The idea of this method is similar to LibSafe (providing a method to check on the boundaries of a pointer before writing to it) but it doesn't perform this check in the start of the vulnerable functions but they provide it as a system to be used by the programmers. The details of the solution are explained in [Speirs 05]

4. Detecting Buffer Overflow security holes in binary files

Since Mores worm 1988, a lot of products and researches have been conducted to protect against buffer overflow attacks. Because the cause of this vulnerability is a lake of checks while developing the applications, most of these products and researches validate the applications source code against this vulnerability. As we have explained in the introduction, analyzing the source code has a lot limitations where they need the source code (which is not always available) and can't be applied by the security administrator. For all these reasons, the paper proposes a new solution for detecting the buffer overflow vulnerabilities in the binary code. As we have explained in section 2, there are two types of buffer overflow vulnerabilities, stack overflow and heap overflow. The proposed solution detects only stack based buffer overflow because it is the most dangerous and common type. Fig. 1 illustrates a general structure for the proposed solution.

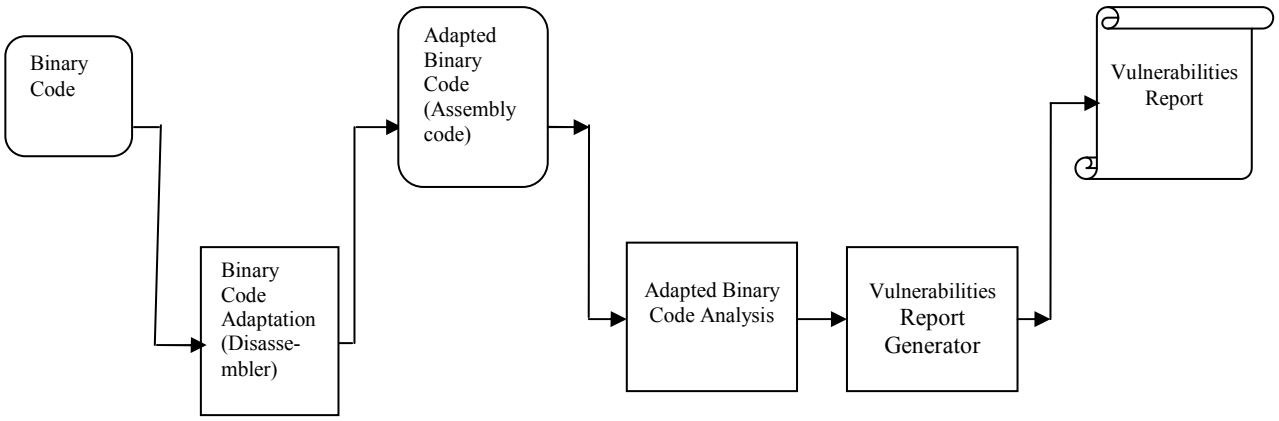

Fig. 1 - Detecting security holes in binary files – general structure

First of all, the binary file must be adapted (transferred into a format which can be analyzed). A lot of approaches are available to adapt the binary files such as disassembling and decompiling. If the binary file can be decompiled into its original source code, the source code static analysis techniques can be applied to detect the vulnerabilities. Unfortunately C/C++ binary de-compilation is not available today. Although the existence of some C (C only not C++) decompilers such as REC [Backer Street Software 2000], DCC [Cifuentes 94] and DisC [Kumar 03], they are not reliable, not accurate and not functioning regarding a lot of C standard code techniques. [CANZANESE 04] explains all the shortcomings and limitations of these de-compilers. After a lot of investigation, we found that the assembly code is the most accurate and suitable format to represent the binary code. After disassembling the binary code, the system analyzes the assembly to detect the buffer overflow vulnerabilities.

5. Stack based buffer overflow security rules

Before explaining the different modules of the proposed model, we will define the security rules which must be followed by any application to be free of stack based buffer overflow security holes. As we have explained before, the stack based buffer overflow attacks occur because of using the vulnerable C functions (strcpy, strncpy, strcat, fgets, gets, getws, sprintf, memcpy, scanf, memmove). As a case study, this section and the following ones explain the security rules and system modules details to detect the vulnerabilities due to using "strcpy" function.

Rule1: The source buffer length must be checked against the destination buffer length

As we have explained before, the stacked based buffer overflow holes occur when a program copies user input buffer into a stack local buffer where the supplied user buffer length is larger than the local buffer length.

The program developer can protect his application, if he checks that the user input buffer length is smaller than the stack buffer length, so we can say that the program

```
# define BUFF_LENGTH 512
Main(int argc, char*[] argv[])
{
        Char buffLocal[BUFF_LENGTH +1];
        If(strlen(argv[1]) < BUFF_LENGTH)  {
        Strcpy(buffLocal, argv[1]) }
}
```

is secure because of the following:

Before using the vulnerable function "strcpy", there is a source buffer length check.

Rule 2: The source buffer length check must be valid

The check of the source buffer length against the destination buffer length must be valid which means that the program must ensure that the source buffer length is less than the destination buffer length.

Rule 3: Any model needs to check that an application is free of buffer overflow holes must validate the application against Rule1 and Rule2

6. Checking the security rules

Checking the validity of an application against Rule1 and Rule2 is not an easy task because of the following:

1- The source buffer may be a parameter passed to the function which performs the actual copy

Really the parameter may be a parameter passed through a chain of functions before performing the copy as illustrated in the following program.

```
void main (int argc, char* SourceBuff)
{ F1 (SourceBuff)  }
// ---- F1 function implementation----------------------------
F1 (char* SourceBuff) {F2 (SourceBuff) ;}
// ---- F2 function implementation ----------------------------
F2 (char* SourceBuff)
{       char DestBuff [256];
        strcpy (DestBuff, SourceBuff);
}
```

2- The source buffer check length may be done through any function of the functions chain

```
void main (int argc, char* SourceBuff)
{If (strlen (SourceBuff) < 256) {F1 (SourceBuff)} }
// ---- F1 function implementation----------
F1 (char* SourceBuff) {F2 (SourceBuff) ;}
// ---- F2 function implementation ----------
F2(char* SourceBuff)
{ char DestBuff [256];  strcpy (DestBuff, SourceBuff); }
```

In this program, although there is no source buffer check length in "F2" function, the check exists in "Main" function. To determine if the program is free of the buffer overflow vulnerability, the checking system must check all the functions chain.

3- Although the existence of a source buffer check length, the validation may not be valid

As illustrated in the following program

```
void main (int argc, char* SourceBuff)
{     int strlen = strlen(SourceBuff)
      If(strlen < 512)  { F1(SourceBuff) ;}
}
// ---- F1 function implementation----------
F1(char* SourceBuff) { F2(SourceBuff); }
// ---- F2 function implementation ----------
F2(char*  SourceBuff)     {char  DestBuff[256];  strcpy
(DestBuff, SourceBuff); }
```

Although this program includes a source buffer check length, the check is not valid because it checks the source buffer length against 512 and while the destination buffer length is 256

4- Although the existence of a valid source buffer check length, the program may include some data operations affecting on the source buffer check length validity

As illustrated in the following program

```
void main (int argc, char* SourceBuff)
{           int sourceBuffLen = strlen(SourceBuff)          .
.           int sourceBuffLen2 = sourceBuffLen -100;
            If(sourceBuffLen2 < 256) { F1(SourceBuff); }
}
F1(char* SourceBuff) { F2(SourceBuff); }
F2(char*     SourceBuff)     {     char     DestBuff[256];
strcpy(DestBuff, SourceBuff);
}
```

The proposed system considers all these security rules. The following section explains the system modules in more details.

7. Buffer Overflow Vulnerability Detection in Binary Code – Design details

As we have explained in section 4, to analyze any binary code generated from C/C++ code, it must be disassemble to generate the corresponding assembly code. Moreover, in section 5, we formed the necessary security rules which must be followed by any application to be free of buffer overflow holes. Furthermore, we illustrated that any system that needs to check the binary applications against buffer overflow holes, it must check these applications against these rules.

Fig. 2 illustrates the details of our proposed system to analyze the assembly code against the buffer overflow security rules.

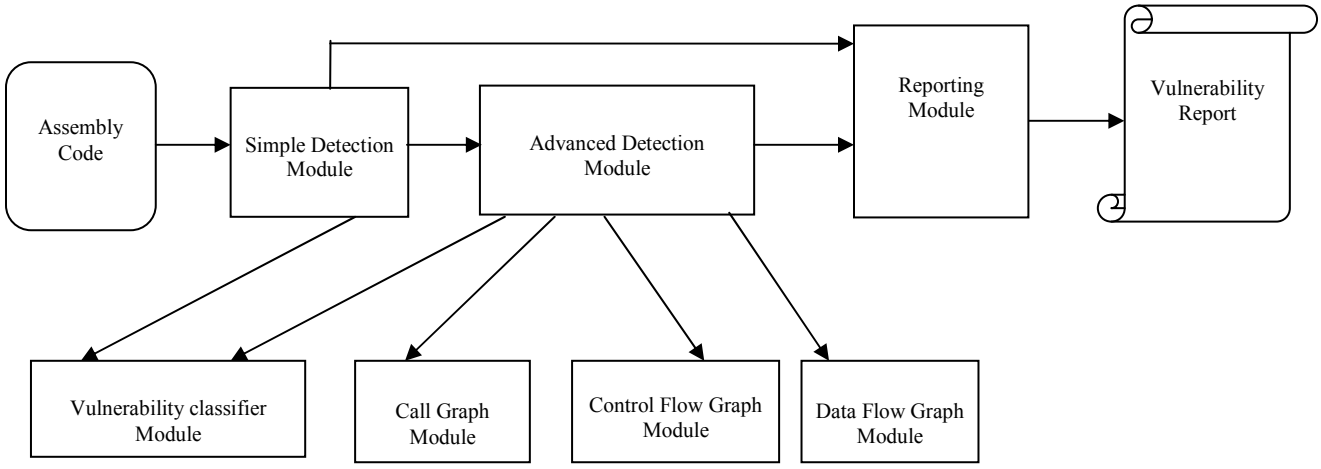

Fig. 2 - Buffer overflow vulnerability detection in binary code – design details

7.1 Simple Detection module

This is the first analysis module to be applied on the assembly file which performs the following:

- Inspects the assembly instructions to detect "strcpy" call.
- Checks if the destination buffer is Stack Local Variable.
- If the source buffer is constant string, it compares the source and destination buffers lengths to detect if this call is vulnerable or not.
- If the source buffer length is larger than the destination buffer length, vulnerability report is generated.
- If the source buffer is string variable, the advanced detection module is called.

Here is a simple code segment for the cases detected by this module

```
Char DestBuffer [10];
strcpy (DestBuffer, "long constant string variable regarding to the destination buffer length");
```

This code segment copies a fixed length string into a stack local variable. Although the source is a constant string, its length is larger than the destination bugger length so the model reports that the application is vulnerable.

7.2 Call Graph module

The call graph is a graph which describes the relationships between a program's procedures. Its nodes represent procedures and its edges represent procedure calls. The Call Graph module is used by the advanced detection module to extract the different execution paths of the application.

7.3 Control Follow Graph module

The Control Follow Graph is a graph which describes the control flow of a program or a program function. The graph is composed of nodes representing the program code segments or functions and edges representing the transition between these nodes. The node is either Normal node which has no Branch Condition or Branch node which has Branch condition. The dominant node of two nodes is the node which exists in all the execution paths between the two nodes. Fig. 3 illustrates a segment of code and its corresponding Control Flow Graph

```
Length = strlen (SourceBuff)
If (Length < 100) {    strcpy (DestBuff, SourceBuff) }
Else {    Message ("Can't copy") }
```

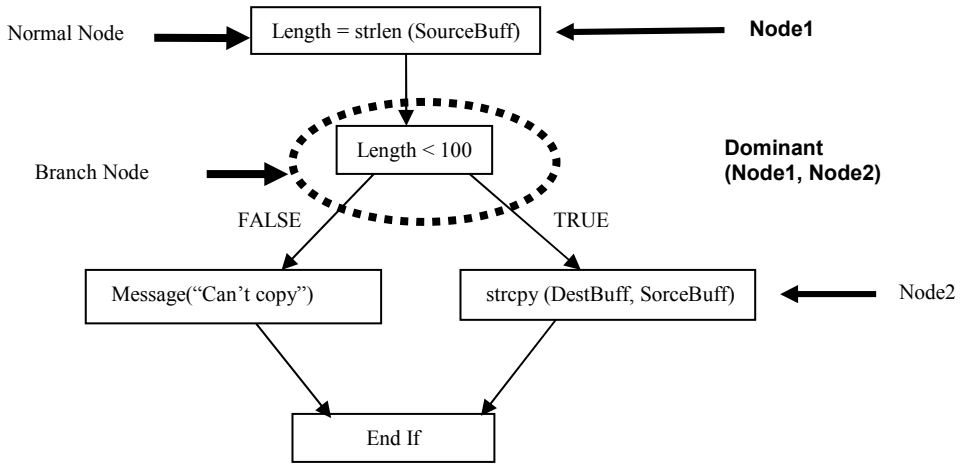

Fig.3 - Control Flow Graph sample

This component is used in conjunction with the Call Graph component by the advanced detection module to check the existence of any dominant node between the node of checking the source buffer length and the node of copying the source buffer.

7.4 Data Flow Graph module

It is a graph representing data dependencies between numbers of operations. From the data flow diagram, the dependence relation between any two variables in the graph can be deduced. This component is used by the advanced detection module to track the operation of any variable.

7.5 Advanced Detection module

The advanced detection module handles the cases in which the source buffer is a character pointer supplied by the user at the application run time which is the most common case and easies the task of the hacker to attack the server through this application.

Here is a simple example for the cases detected by this module

```
strcpy (DestBuffer,
SourceBuffer)
```

where ***SourceBuffer*** is supplied by the user to the application and ***DestBuffer*** is stack local variable

The main task of the module is to check that ***SourceBuffer*** length is checked safely against ***DestBuffer*** length before "strcpy" call.

This task is not easy as we have explained in details in section 5 where:

- The Source buffer length may be checked in the strcpy function caller or any caller to this caller and so on a long the execution path.
- The result of the source buffer length may not be referenced (not checked before strcpy call).
- The SourceBuffer check length variable may be changed after or through the check length code segment and before strcpy call code segment.

To solve all these problems and others, the advanced detection module briefly does the following:

- Calculates all the execution paths of strcpy function caller by using the Call graph module.
- Analyzes all these paths to detect at least one vulnerable path by using the Control Flow graph and Data Flow Graph modules in addition to its own analysis for the retrieved data from these two modules.

7.6 Reporting module

This module is responsible for generating vulnerabilities report which accurately describes the vulnerability and how to remove it. The report includes the name of the vulnerable function, the name of the immediate vulnerable function caller, the cause of the vulnerability, the class of the vulnerability, how the hacker can exploit the vulnerability and how to remove it.

7.7 Vulnerability classifier module

To generate an accurate vulnerability report, the vulnerabilities are classified according to their dangerous from security point of view. Two parameters determine the class of the vulnerability, the source buffer type (SBT) and the source buffer data flow equation (SBE) which describes the relation between the check length variable and dominant node branch condition left hand side. Table 1 enumerates the different combinations of these variables and the corresponding vulnerability class.

SBT	SBE	Vulnerability class
Constant	*	Low Dangerous
User Input	Invalid Equation	High Dangerous
User Input	Undecided Equation	Dangerous

Table 1 Buffer overflow vulnerability classes

Let's read the rows of the table. The first row tells that, if the source buffer type is constant buffer and regardless the source buffer equation, the vulnerability type is low dangerous. The vulnerability is classified as Low Dangerous because it is unattended programming error and its maximum influence is to crash the application. The second row tells that, if the source buffer is user input data and the source buffer equation is invalid (which means that the source buffer check length is not secure against the destination buffer length), the vulnerability is High Dangerous. The vulnerability is high dangerous because it is a big hole in the application where there is an input buffer from the user which is copied to a stack local variable with invalid check length. This vulnerability can be exploited by hackers to overflow the local variable. The third row tells that, if the source buffer is user input data and the source buffer equation is Undecided, the vulnerability is Dangerous because it is not an absolute hole.

7.8 Dominant Node's Branch Condition Analysis

The branch condition of the dominant node plays a significant rule in determining if the application is vulnerable or not. Here is an example for the branch condition of the dominant node

```
if (SourceLen < 256)
```

Based on this condition and the value of the Transition Edge between the Dominant Node and strcpy node (or any function in the execution path of strcpy function) (TRUE or FALSE), strcpy function is called. The advanced detection module analyzes this condition in conjunction with the Transition Edge value. Based on this analysis, the advanced detection module generates two security conditions. If any one of these conditions is not satisfied, the application is vulnerable.

Table 2 illustrates all the different combinations of the Branch Condition in conjunction with the Transition Edge value and the corresponding security conditions.

Branch condition	Transition Edge	Security conditions
LHS < RHS	TRUE	SourceBuffLen <= LHS DestBuffLen >= RHS
LHS < RHS	FALSE	SourceBuffLen >= LHS DestBuffLen <= RHS
LHS > RHS	TRUE	SourceBuffLen <= RHS DestBuffLen >= LHS
LHS > RHS	FALSE	SourceBuffLen >= RHS DestBuffLen <= LHS
LHS = RHS	TRUE	SourceBuffLen <= LHS DestBuffLen >= RHS
LHS = RHS	FALSE	Undecided

Table 2 Dominant node branch condition analysis

Note: LHS stands for the left hand side of the branch condition while RHS stands for the right hand side of the branch condition.

8. Buffer Overflow Vulnerability Detection in Binary Code – System Implementation

8.1 System modules implementation packages

As we have explained in section 5, the binary code is disassembled and then the generated assembly code is analyzed to detect the buffer overflow vulnerabilities. Here is a list of the system modules and how we have implemented each one of them.

1- **Binary adaptation module (Disassembler):** We have used IDA Pro Disassembler [IDA Pro Web Site]
2- **Simple Detection module:** It is written as IDC script. IDC is the easiest way for programming IDA Pro. It is a scripting language which is very similar to ANSI C language.
3- **Advanced Detection module:** IDC script.
4- **Vulnerability Classifier module:** IDC script.
5- **Call Graph:** IDC script.
6- **Control Flow Graph:** It is written as IDA plug-in**.** In case the required code exceeds the capabilities of IDC, we can resort to writing an IDA plug-in. IDA Pro Plug-ins are developed in C++ and can call IDA Pro APIs directly.
7- **Data Flow Graph:** IDA Plug-in.

8.2 IDC Script Sample

Listing all the scripts and Plug-ins code is out of scope of this paper but we will list the IDC script of one of the **simple detection module** functions.

The function is called **FindVulnerableFunction2** and it is used to find a vulnerable function call in an assembly file stating from a specific address.

```
static FindVulnerableFunction2(vulFuncName, startAddress)
{auto tempAddress;
        while(startAddress != -1)
        {
        if(GetMnem(startAddress) == "call")
                {
                if(GetOpnd(startAddress, 0) == vulFuncName)
                {return startAddress; }
                }
        startAddress = NextAddr(startAddress);
        }
        return startAddress;
}
```

8.3 Detecting vulnerable application example

In this section we will illustrate a vulnerable application and explain the rule of each module to detect the application vulnerability. The following program is a very complicated vulnerable program where it has a non-valid source buffer check length due to modification in the source buffer check length variable "sourceBuffLen"

```
        void main (int argc, char* SourceBuff)
        {
                int sourceBuffLen = strlen(SourceBuff)
                int sourceBuffLen2 = sourceBuffLen -100;
                If(sourceBuffLen2 < 256) { F1(SourceBuff); }
        }
        F1(char* SourceBuff)
        { F2(SourceBuff); }

        F2(char* SourceBuff)
        {       char DestBuff[256];
                strcpy(DestBuff, SourceBuff);
        }
```

Fig. 4 illustrates the rule of each module in the system to detect the security hole of this application.

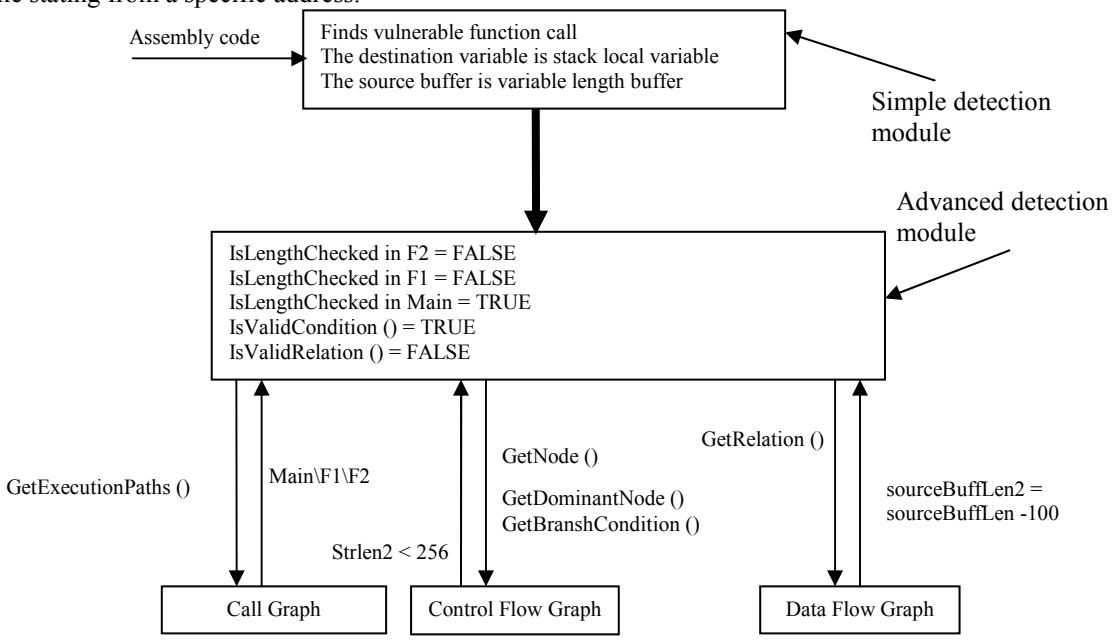

Fig. 4 – The rule of each module to detect a vulnerable application

Conclusion

A lot the internet servers applications contain several security holes which enable the hackers to attack these servers although the servers are protected by Firewalls, Intrusion Detection Systems and other security techniques. Early detecting the applications' security holes increases the applications reliability and aids security administrators to protect their critical servers.

In this paper, we have approved that the applications security holes can be detected in the applications binary format. We have provided a general model for detecting the applications security holes, and then we have provided the model details to detect stack based buffer overflow security holes.

Also we have provided the security rules the applications must follow to avoid the existence of buffer overflow holes and explained how the proposed model verifies the applications against these rules.

References

[Backer Street Software 2000] http://www.backerstreet.com/rec/rec.htm, 1997 – 2005

[CANZANESE 04] RAYMOND J. CANZANESE, JR., MATTHEW OYER, SPIROS MANCORIDIS, and MOSHE KAM, " A Survey of Reverse Engineering Tools for the 32-Bit Microsoft Windows Environment ", ACM Journal Name, Vol. V, No. N, Month 20YY, 2004

[Cifuentes 94] http://www.itee.uq.edu.au/ristina/dcc.html

[Chess 04] Brian Chess and Gary McGraw, "Static Analysis for Security", IEEE SECURITY & PRIVACY, 2004

[Chen 02] "MOPS: An Infrastructure for Examining Security Properties of Software," Proc. 9th ACM Conf. Computer and Communications Security (CCS2002), ACM Press, 2002, pp. 235–244.

[Cowan 2000] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole, "Buffer Overflows - Attacks and Defenses for the Vulnerability of the Decade" In DARPA Information Survivability Conference and Expo 2000.

[FAYOLLE 02] Pierre-Alain FAYOLLE, Vincent GLAUME, "A Buffer Overflow Study Attacks & Defenses", International Conference on Dependable Systems and Networks (DSN'02) Washington, D.C., USA, June 23 - 26, 2002

[Ghosh 98] Anup K Ghosh, Tom O'Connor, and Gary McGraw, "An Automated Approach for Identifying Potential Vulnerabilities in Software", In the Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, May 1998.

[IDA Pro Web Site] http://www.datarescue.com/

[Kumar 03] http://www.debugmode.com/dcompile/disc.htm

[Simon 01] Istvan Simon, "A Comparative Analysis of Methods of Defense against Buffer Overflow Attacks", California State University, Hayward Hayward, CA 94542, January, 31 2001

[Speirs 05] William R. Speirs. "Making The Kernel Responsible: A New Approach To Detecting & Preventing Buffer Overflows", The Advanced Technology Research Center, 2005.

[Tsai 02] Timothy Tsai and Navjot Singh, "Libsafe: Transparent System-wide Protection Against Buffer Overflow Attacks", International Conference on Dependable Systems and Networks (DSN'02) Washington, D.C., USA, June 23 - 26, 2002

[Wagner 2000] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities" In NDSS (Network and Distributed System Security), San Diego, CA, February 2000.

[Viega 2000] John Viega, J.T. Bloch, Yoshi Kohno and Gary McGraw, "ITS4: A Static Vulnerability Scanner for C and C++ Code", Reliable Software Technologies, Dulles, Virginia