RICB: Integer Overflow Vulnerability Dynamic Analysis via Buffer Overflow

Yong Wang[1,2], Dawu Gu[2], Jianping Xu[1], Mi Wen[1], and Liwen Deng[3]

[1] Department of Compute Science and Technology,
Shanghai University of Electric Power, 20090 Shanghai, China
[2] Department of Computer Science and Engineering,
Shanghai Jiao Tong University, 200240 Shanghai, China
[3] Shanghai Changjiang Computer Group Corporation, 200001, China
wy616@126.com

Abstract. Integer overflow vulnerability will cause buffer overflow. The research on the relationship between them will help us to detect integer overflow vulnerability. We present a dynamic analysis methods RICB (Run-time Integer Checking via Buffer overflow). Our approach includes decompile execute file to assembly language; debug the execute file step into and step out; locate the overflow points and checking buffer overflow caused by integer overflow. We have implemented our approach in three buffer overflow types: format string overflow, stack overflow and heap overflow. Experiments results show that our approach is effective and efficient. We have detected more than 5 known integer overflow vulnerabilities via buffer overflow.

Keywords: Integer Overflow, Format String Overflow, Buffer Overflow.

1 Introduction

The integer overflow occurs when positive integer changing to negative integer after addition or an arithmetic operation attempts to create a numeric value that is larger than that can be represented within the available storage space. It is old problem, but now faces the security challenge once the integer overflow vulnerabilities are used by hackers. The number of integer overflow vulnerabilities has been increasing rapidly in recent years. With the development of the vulnerabilities exploit technology, the detection methods of integer overflow are made rapid growth.

The IntScope is a systematic static binary analysis tools. It is based approach to particularly focus on detecting integer overflow vulnerabilities. The tool can automatically detect integer overflow vulnerabilities in x86 binaries before an attacker does, with the goal of finally eliminating the vulnerabilities [1]. Integer overflow detection method based on path relaxation is described for avoiding buffer overflow through lightly static program analysis. The solution traces the key variables referring to the size of a buffer allocated dynamically [2].

The methods or tools are classified into two categories: static source code detection and dynamic running detection. Static source code detection methods are composed of IntScope[1], KLEE[3], RICH[4], EXE[5], and the dynamic SAGE[12].

X. Lai et al. (Eds.): E-Forensics 2010, LNICST 56, pp. 99–109, 2011.

KLEE is a symbolic execution tool, which is capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs [3]. RICH (Run-time Integer Checking) is a tool for efficiently detecting integer-based attacks against C programs at run time [4]. EXE works well on real code, finding bugs along with inputs that trigger them, which runs it on symbolic input initially [5]. The SAGE (Scalable, Automated, Guided Execution) is a tool employing x86 instruction-level tracing and emulation for white box fuzzing of arbitrary file-reading windows applications [12].

Integer overflow can cause string format overflow, buffer overflow such as stack overflow and heap overflow. CSSV (C String Static Verify) is a tool that statically uncovers all string manipulation errors [6]. FormatGuard is an automatic tools for protection from printf format string vulnerabilities [13]. Buffer overflows in C program language occur easily because C provides little syntactic checking of bounds [7]. Besides static analysis tools, the dynamic buffer overflow analysis tools are used in the detection. Through comparison among tools publicly available for dynamic buffer overflow prevention, we can value the dynamic intrusion prevention efficiently [8].

Research on relationship between the buffer overflow and string format overflow can help us to reveal the buffer overflow internal features [9]. There are some applications such as integer squares with overflow detection [10] and integer multipliers with overflow detection [11].

Our previous related research is focusing on denial of service detection [14] and malicious software behavior detection [15]. The integer overflow vulnerability research can help us to reveal the malware intrusion procedure by exploiting overflow vulnerability to execute shell code. The key idea of our approach is dynamic analysis on the integer overflow via (1) format string overflow; (2) stack overflow; (3) heap overflow.

Our contributions include:

(1) We propose a dynamic method of analyzing the integer overflow via buffer overflow.
(2) We present analysis methods of the buffer overflow interruption change procedure which is caused by integer overflow.
(3) We implement the methods and experiments show that they are effective.

2 Integer Overflow Problem Statement

2.1 Signed Integer and Unsigned Integer Overflow

The register width of a processor determines the range of values that can be represented. Typical binary register widths include: 8 bits, 16 bits, 32 bits. The CF (Carry Flag) and OF (Overflow Flag) in PSW (Program Status Word) represent signed and unsigned integer overflow, respectively. The details are shown in Table 1:

When CF and OF equal to 1, the signed or unsigned integer overflow. If CF=0 and OF=1, the signed integer overflows. If CF=1 and OF=0, the unsigned integer overflow. The integer memory structure is described in Fig. 1, when it overflows.

Table 1. Types and examples of integer overflow

Type	Width	Boundary	Overflow Flag	
char	8 bits	0~255	CF=1	OF=1
Signed Short	16 bits	-32768~32767	CF=0	OF=1
Unsigned Short	16 bits	0 ~ 65535	CF=1	OF=0
Signed Long	32 bits	-2,147,483,648 ~ 2,147,483,64	CF=0	OF=1
Unsigned Long	32 bits	0 ~ 4,294,967,295	CF=1	OF=0

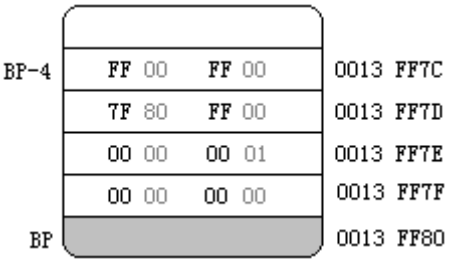

Fig. 1. Integer overflow is composed of signed integer overflow and unsigned integer overflow. The first black column is the signed integer 32767 and the first gray column is -32768. The second black column is the unsigned integer 65535 and the second gray column is 0.

2.2 Relationship between Integer Overflow and Other Overflow

The relation between the integer overflow and other overflows such as string format overflow, stack overflow and heap overflow is shown in formula 1:

$$\begin{cases} \left\{ OV \middle| OV_{Integer} \wedge OV_{StringFormat} \wedge OV_{Stack} \wedge OV_{Heap} \right\} \subset OverFlow \\ \left\{ OV_{stringFormat} \wedge OV_{Stack} \wedge OV_{Heap} \right\} \bigcap OV_{Integer} \neq \varnothing \end{cases} \tag{1}$$

The first line in formula (1) means that overflows include integer overflow, string format overflow, stack overflow and heap overflow. The last line in formula (1) means that the integer overflow can cause the other overflow.

The other common overflow types and examples caused by integer overflow are located some special format string or functions, which are listed in Table 2:

Table 2. Overflow types and examples caused by integer overflow

Integer Overflow Type	Boundary	Examples
Format String Overflow	Overwrite memory	printf("format string %s %d %n", s,i);
Stack Overflow	targetBuf < sourceBuf	memcpy(smallBuf, largeBuf, largeSize)
Heap Overflow	heapSize < largeSize	HeapAlloc(hHeap, 0,largeSize)

In Table 2, if the integer in format strings, stack and heap overflow, the integer overflow can cause the corresponding types overflow.

2.3 Problem Scope

In this paper, we focus on the relationship between the integer overflow and the other overflow such as format string overflow, stack overflow, and the heap overflow.

3 Dynamic Analysis via Buffer Overflow

3.1 Format String Overflow Exploitation Caused by Integer Overflow

Format string overflow is one kind of Buffer overflow in some sense. In order to print program results on the screen, program needs to use the printf () function in C language. The function has two types of parameters: format control parameters and output variables parameters. The format control parameters are composed of string format %s, %c, %x, %u and %d. The out variables parameters types may be integer, real, string or address pointer. The common used format string program is presented as below:

```
char *s="abcd";
int i=10;
printf("%s %d",s,i);
```

Char pointer s stores the string address and integer variables I has its initial value 10. Printf () function uses the string format parameters to define the output format. The printf () function will use stack to store its parameters. The printf () has three parameters: the format control string pointer pointing to the string "%s %d", the string pointer variable pointing to the string "abcd" and integer variable I with initial value 10.

String contents can store assembly language instruction by \x format. For instance if the hexadecimal code of assembly language instruction "mov ax,12abH" is B8AB12H, then the shellcode is "\xB8\xAB\x12". When the IP points to the shellcode memory contents, the assembly language instructions will be executed.

The dynamic execute procedure of the program is shown in Fig. 2

Format string will overflow, when data is beyond the string boundary. The vulnerabilities can be used to crash a program or execute the harmful shell code by hacker. The problem exits the C language function, such as printf ().

The malicious may use the parameters to overwrite data in the stack or other memory locations. The dangerous parameter %n in ANSI standard, by which you can write arbitrary data to arbitrary location, is disabled by default in Visual Studio 2005.The following program will make format string overflow.

```
int main(int argc, char *argv[])
{
    char *s="abcd";
    int i=10;
    printf("\x10\x42\x2f\x3A%n",s,i,argv[1]);
    return 0;
}
```

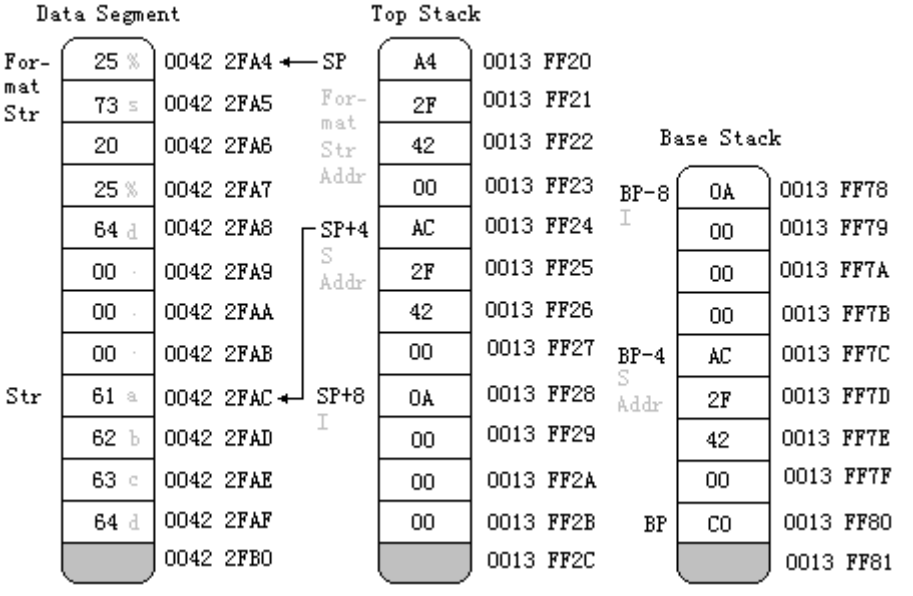

Fig. 2. String Format printf("%s %d", s, i) has three parameters: the format string pointer SP, the s string pointer SP+4, and the integer i saved in 0013FF28H memory address. The black hexadecimal numbers in the box are the memory values. The black side hexadecimal numbers are the memory address.

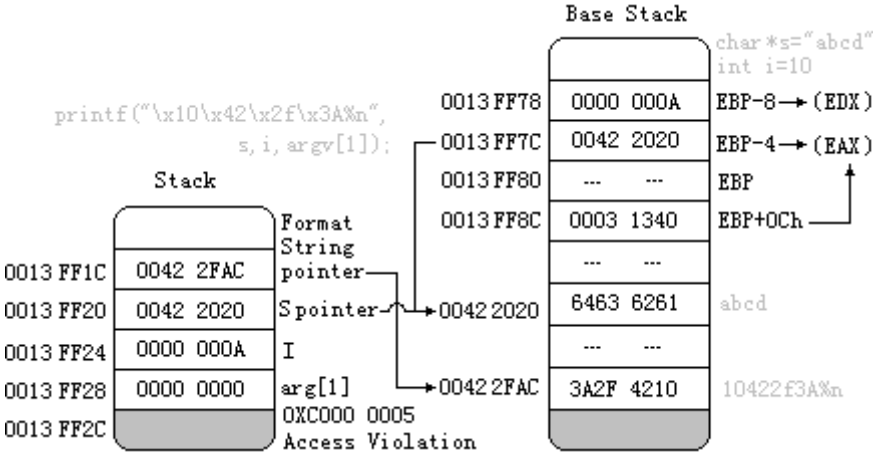

Fig. 3. Format string overflowed at 0XC0000005 physical address. When the char and integer variable are initialed, the base stack memory is shown on the left side. When the printf () function is executed, the stack changing procedure is described on the left side. The first string format control parameter in memory 00422FAC address, the second parameter S pointer to the 00422020 address. Integer variable I and argv[1] pointer are pushed into the stack firstly.

The main function has two parameters: integer variable argc and char integer variable argv[]. If the program executes in console command without input arguments, the argc equals to 1 and the argv[1] is null. The argv[1] is integer down overflow. The execute procedure of the program in stack and base stack memory is shown in Fig.3:

3.2 Stack Overflow Exploitation Caused by Integer Overflow

Stack overflow is the main kind of buffer overflow. As the strcpy () function has not bounds checking, once the source string data beyond the target string buffer bounds and overwrite the function return address in stack buffer, the stack overflow will occur. The integer upper or down overflow will also cause stack overflow. The example program is as shown as bellow.

```
int stackOverflow (char *str)
{
     char buffer[8]="abcdefg";
     strcpy(buffer,str);
     return 0;
}

int main(int argc, char *argv[])
{
     int i;
     char largeStr[16]="12345678abcdefg";
     char str[8]="1234567";
      stackOverflow(str);
     stackOverflow(largeStr);
     stackOverflow(argv[1]);
}
```

The function calling procedure mainly includes six main steps:

(1) The real parameters of called function are pushed into stack from right to left. The example real parameter string address is pushed into stack.
(2) Push instruction: call @ILT+5(stackOverflow) (0040100a) next IP address (00401145) into stack.
(3) Push EBP address into stack; EBP new value equals to ESP by instruction: Mov EBP,ESP; Create new stack space for sub function local variables by instruction: Sub ESP,48H.
(4) Push EBX, ESI, EDI into stack.
(5) Move offset of [EBP-48H] to EDI; Copy 0CCCCCCCCH to DWORD[EDI]; Store local variables in sub function to [EBP-8] and [EBP-4].
(6) POP local variables and return.

The memory change procedure is presented in Fig. 4 during the main function calling the stack overflow sub function.

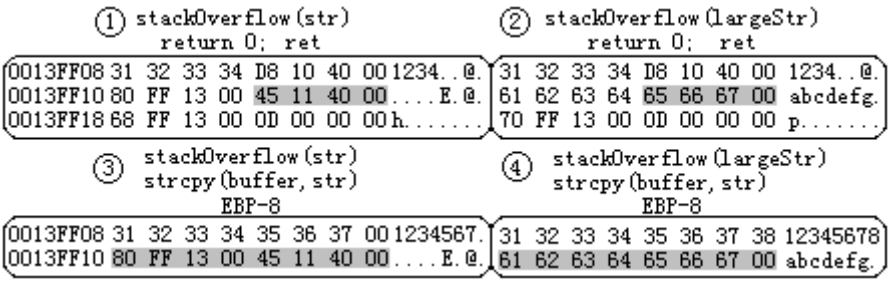

Fig. 4. Stackoverflow(str) return address is 00401145H as shown in figure (1); StackOverflow (largest) return address is 00676665H as shown in figure (2); Base stack memory status of [EBP-8] after strcpy(buffer,str) with str parameter is shown in figure (3); with largest parameter is shown in figure (4).

The access violation is derived from the large string upper integer overflow and argv[1] down integer overflow. The stack overflow caused by integer overflow break the program at the physical address 0xC0000005.

Once the return address content in stack is overwritten by stack buffer overflow or integer overflow, the IP will jump to the overwrite address. If the address points to the shell code, which is the malicious code for intruding or destroying computer system, the original program will execute the malicious shell code. Many kinds of shell codes can be got from shellcode automatic tools.

It is difficult to dynamically locate the overflow instruction physical location. Once finding the location point, you can overwrite the jump instruction into the overflow point. Getting the overflow point has two methods: manually testing methods and insert assembly language. The inserted key assembly language in the front of the return function is: lea ax, shellcode; mov si,sp; mov ss:[si],ax.

The other locating overflow point method is manually testing shown in Table 3:

Table 3. Locate the overflow address point caused by integer upper overflow

Disassembly code	Register value befor running	Register value after running
xor eax,eax	(eax)=0013 FF08H	(eax)=0000 0000H
pop edi	(edi)= 0013 FF10H	(edi)= 0013 FF80H
pop esi	(esi) = 00CF F7F0H	(esi)= 00C FF7F0H
pop ebx	(ebx)=7FFD 6000H	(ebx) =7FFD 6000H
add esp,48h	(esp)= 0013 FEC8H	(esp)= 0013 FF10H
cmp ebp,esp	(ebp)=(esp)= 0013 FF10H	(ebp)=(esp)= 0013 FF10H
call _chkesp	(esp)= 0013 FF10H	(esp) = 0013 FF0CH
ret	(esp) = 0013 FF0CH	(esp)=0013 FF10H
mov ebp,esp	(ebp)=(esp)= 0013 FF10H	(ebp)=(esp)= 0013 FF10H
pop ebp	(ebp)=(esp)= 0013 FF10H	(ebp) = 6463 6261H
ret	(eip) = 0040 10DBH	(eip)= 0067 6655H

3.3 Heap Overflow Exploitation Caused by Integer Overflow

Heap overflow is another important type of buffer overflow. Heap has different data structure from stacks. Stack is FILO (First In Last Out) data structure, which is always used in function calling. Heap is a memory segment that is used for storing dynamically allocated data and global variables. The functions of creating, allocating and free heap are HeapCreate (), HeapAlloc() and HeapFree().

Integer overflow can lead to heap overflow, when the memory addresses are overwritten. The argv[0] is a string pointer. atoi(argv[0]) equals to 0. If the atoi(argv [0]) is the HeapAlloc() function last parameter, It will lead to integer overflow. The program is presented as bellow:

```
int main(int argc, char *argv[])
{
     char *pBuf1,*pBuf2;
     HANDLE hHeap;
     char myBuf[]="intHeapOverflow";
     hHeap=HeapCreate(HEAP_GENERATE_EXCEPTIONS,
                                        0X1000,0XFFFF);
     pBuf1=(char *)HeapAlloc(hHeap,0,8);
     strcpy(pBuf1,myBuf);
     pBuf2=(char *)HeapAlloc(hHeap,0, atoi(argv[0]));
     strcpy(pBuf2,myBuf);
     HeapFree(hHeap,0,pBuf1);
     HeapFree(hHeap,0,pBuf2);
     return 0;
}
```

The program defines two buffer pointers: pBuf1 and pBuf2 and creates a heap with the return hHeap pointer. The variables and heap structure in memory is shown in Fig. 5:

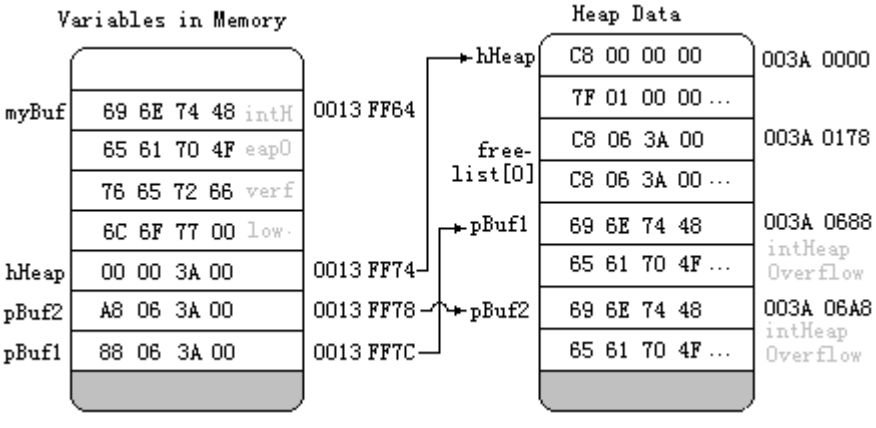

Fig. 5. Variables in memory are shown in left and heap data are in the right. Handle pointer hHeap save heap address. The heap variables pointers pBuf1 and pBuf2 point to their corresponding data in the heap. String variables myBuf save in 0013FF64 address.

The heap next and previous addresses in free list are shown as Fig. 6:

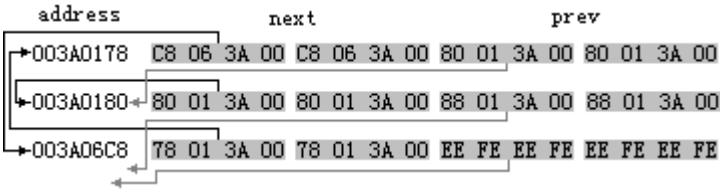

Fig. 6. In the free double link list array, there are next pointer and previous pointer. When allocating a dynamic memory using HeapAlloc () function, a heap free space will be used. Heap overflow will occur if the double link list are destroyed by overwritten string caused by integer overflow.

The program occurs heap overflow which is caused by integer overflow at the IP address 7C92120EH. The integer overflow includes the situation that size of mybuf and is larger than myBuf1 and myBuf2. The max size of myBuf2 allocation is zero as a result of atoi(argv[1]).

4 Evaluation

4.1 Effectiveness

We have applied RICB to analyze integer overflow with format string overflow, stack overflow, heap overflow. RICB methods successfully dynamically detected the integer over flow in examples, and also find the relationship between the integer overflow and buffer overflow.

As RICB is a dynamic analysis method, it may face the difficulties from static C language. To confirm the suspicious buffer overflow vulnerability is really caused by integer overflow, we rely on our CF (Carry Flag) and OF (Overflow Flag) in PSW (Program Status Word).

4.2 Efficiency

The RICB method includes the following steps: decompiling execute file to assembly language; debug the execute file step into and step out; locate the over flow points; check analysis integer overflow via buffer overflow. We measure the three example program on a Intel (R) Core (TM)2 Duo CPU E4600 (2.4GHZ) with 2GB memory running Windows. Table 4 shows the result of efficiency evaluation.

Table 4. Evaluation result on efficiency

File Name	Overflow EIP	Access Violation	Integer Overflow
FormatString.exe	0040 1036	0XC000 0005	argv[1] %n
Stack.exe	0040 1148	0XC000 0005	argv[1] largeStr
Heap.exe	7C92 120E	0X7C92 120E	atoi(argv[0])

5 Conclusions

In this paper, we have presented the use of RICB methods to dynamical analysis of run-time integer checking via buffer overflow. Our approach includes the steps: decompiling execute file to assembly language; debug the execute file step into and step out; locate the over flow points; check analysis buffer overflow caused by integer overflow. We have implemented our approach in three buffer overflow types: format string overflow, stack overflow and heap overflow. Experiment results show that our approach is effective and efficient. We have detected more than 5 known integer overflow vulnerabilities via buffer overflow.

Acknowledgments. The work described in this paper was supported by the National Natural Science Foundation of China (60903188), Shanghai Postdoctoral Scientific Program (08R214131) and World Expo Science and Technology Special Fund of Shanghai Science and Technology Commission (08dz0580202).

References

1. Wang, T.L., Wei, T., Lin, Z.Q., Zou, W.: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In: Proceedings of the 16th Network and Distributed System Security Symposium, San Diego, CA, pp. 1–14 (2009)
2. Zhang, S.R., Xu, L., Xu, B.W.: Method of Integer Overflow Detection to Avoid Buffer Overflow. Journal of Southeast University (English Edition) 25, 219–223 (2009)
3. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008), San Diego, CA (2008)
4. Brumley, D., Chiueh, T.C., Johnson, R., Lin, H., Song, D.: Rich: Automatically Protecting Against Integer-based Vulnerabilities. In: Proceedings of the 14th Annual Network and Distributed System Security Symposium, NDSS (2007)
5. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: Automatically Generating Inputs of Death. In: Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, pp. 322–335 (2006)
6. Dor, N., Rodeh, M., Sagiv, M.: CSSV: Towards a Realistic Tool for Statically Detecting all Buffer Overflows. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, San Diego, pp. 155–167 (2003)
7. Haugh, E., Bishop, M.: Testing C Programs for Buffer overflow Vulnerabilities. In: Proceedings of the10th Network and Distributed System Security Symposium, NDSS SanDiego, pp. 123–130 (2003)
8. Wilander, J., Kamkar, M.: A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In: Proceedings of the 10th Network and Distributed System Security Symposium, NDSS 2003, SanDiego, pp. 149–162 (2003)
9. Lhee, K.S., Chapin, S.J.: Buffer Overflow and Format String Overflow Vulnerabilities, Sofware-Practice and Experience, pp. 1–38. John Wiley & Sons, Chichester (2002)
10. Gok, M.: Integer squarers with overflow detection, Computers and Electrical Engineering, pp. 378–391. Elsevier, Amsterdam (2008)

11. Gok, M.: Integer Multipliers with Overflow Detection. IEEE Transactions on Computers 55, 1062–1066 (2006)
12. Godefroid, P., Levin, M., Molnar, D.: Automated whitebox fuzz testing. In: Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA (2008)
13. Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G.: FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In: Proceedings of the 10th USENIX Security Symposium. USENIX Association, Sydney (2001)
14. Wang, Y., Gu, D.W., Wen, M., Xu, J.P., Li, H.M.: Denial of Service Detection with Hybrid Fuzzy Set Based Feed Forward Neural Network. In: Zhang, L., Lu, B.-L., Kwok, J. (eds.) ISNN 2010. LNCS, vol. 6064, pp. 576–585. Springer, Heidelberg (2010)
15. Wang, Y., Gu, D.W., Wen, M., Li, H.M., Xu, J.P.: Classification of Malicious Software Behaviour Detection with Hybrid Set Based Feed Forward Neural Network. In: Zhang, L., Lu, B.-L., Kwok, J. (eds.) ISNN 2010. LNCS, vol. 6064, pp. 556–565. Springer, Heidelberg (2010)