Memory-Size-Assisted Buffer Overflow Detection

Chunguang Kuang*
Science and Technology on Information System Security Laboratory, Beijing Institute of System Engineering, Beijing, China
Email: cpprc2@263.net

Chunlei Wang and Minhuan Huang
Science and Technology on Information System Security Laboratory, Beijing Institute of System Engineering, Beijing, China
Email: david.zou.01@gmail.com, white1white1@163.com

***Abstract*--Since the first buffer overflow problem occurred, many detection techniques have been presented. These techniques are effective in detecting most attacks, but some attacks still remain undetected. In order to be more effective, a memory-size-assisted buffer overflow detection(MBOD) is presented. The key feature of buffer overflow is that the size of the source memory is bigger than the size of the destination memory when memory copying operation occurs. By capturing memory copying operation and comparing memory size at run time, MBOD detects buffer overflow. MBOD collects the information of memory size in both dynamic way and static way. An implementation shows that the technique is feasible.**

***Index Terms*--buffer overflow, detection, memory size, stack, heap**

I. Introduction

There are many kinds of buffer overflow attack [1]. The most common one is overwriting the return address on the stack with the address of a piece of malicious code. When the RET instruction(return instruction in binary code) in the victim function is executed, program control is transferred to the malicious code. This is the stack smashing attack [2].

The less common form of buffer overflow attack is memory pointer attack [3]. It corrupts memory pointer variables on the stack, which also causes program control transferred.

The third form of buffer overflow attack is frame pointer attack. It overwrites the old base pointer on the stack with the address of a piece of malicious code. When the caller function returns, the program control is transferred to the malicious code.

Other less harmful buffer overflow attacks overwrite heap buffers and static buffers.

The form of buffer overflow is diverse. If a detection technique is based on the form, the detection technique can not be effective in detecting all forms of buffer overflow. Fortunately, there is a key feature in all forms of buffer overflow. The key feature of buffer overflow is that the size of the source memory is bigger than the size of the destination memory when memory copying operation occurs. Our technique is based on this feature, and our work is as follows,

1) Design a detection technique.

2) Implement a detector. The detector is a Pintool built on the Pin [4]. Binary code instead of source code of an application is required. The detector tries to detect the possible buffer overflow rather than prevent a triggered off buffer overflow from advancing, therefore, it can find vulnerabilities in programs even when the tested data does not trigger off an overflow. The detection process needs to be executed only once, not to be executed whenever the application is executed, so the overhead is low. It detects all forms of buffer overflow.

The rest of the paper is organized as follows: Section 2 presents the related works in the area of buffer overflow detection. Section 3 presents detection technique of MBOD(memory-size-assisted buffer overflow detection). Section 4 describes implementation of MBOD. Section 5 summarizes and discusses the result. Section 6 concludes our work.

II. Related Work

Austin et al. provide a pointer and array access checking technique [5]. C pointers are transformed into safe pointers. Safe pointer contains fields such as the base address, its size and its scope. Access check uses these fields to determine whether the pointer is safe. Source code is needed.

Jones and Kelly present an array bounds and pointer checking technique [6]. A table of all the valid storage objects is maintained. Storage object has information such as the base address and size etc. Heap object information is collected by modified functions of malloc() and free(). Stack object information is collected by constructor and destructor functions. These objects

Manuscript received March 2, 2013; revised June 15, 2013; accepted June 27, 2013.

doi:10.4304/jsw.9.2.336-342

information are used for access when array and pointer operations occur. The technique is implemented as an extension of GNU C compiler. Source code is needed.

Solar Designer's Linux kernel patch makes the stack segment non-executable [7]. Thus the malicious code that stays in the stack can not be executed. It is required to change the operating system.

StackGuard [8] is effective in protecting return address from being altered. StackGuard is a GNU C compiler extension. It inserts canary word between the return address and the variables when programs are compiled. If an attacker tries to overwrite the return address by overflowing the variables, the canary word must be overwritten. So StackGuard can detect any attempt to alter the return address by inspecting the canary word before returning from a function. The inspecting process must be executed at every execution, and source code is needed to recompile the program.

Libsafe [9, 10] builds a new C library to replace the standard C library. The new C library is preloaded to intercept C library function calls. In the new C library, the vulnerable copy functions such as strcat() are changed. In addition to the original functionality, these functions are imposed more functionality. They can control them not to overwrite return address. The based idea is that the buffer can not extend beyond its stack frame. So the maximum size of a buffer is the distance between the address of the buffer and the frame pointer. It means copying operation can not overwrite the frame pointer, and thus copying operation can not overwrite the return address that is more away from the buffer than the frame pointer is. Source code is not needed unless the program is statically linked with C library. But the protecting process must be executed at every execution.

An integer range analysis technique by Wagner et al. [11] models string buffer on a pair of integer ranges. A set of integer constraints is predefined for a set of string operations. It makes sure that every string operation not to overwrite the nearby memory space. Source code is needed, and it is only for C vulnerable functions.

An annotation-assisted static analysis technique by Larochelle and Evans based on LCLint uses annotations [12]. Programmers are required to add annotations to vulnerable functions. It is a burden for programmers to do the annotation work.

StackShield [13] is similar to StackGuard. StackShield is efficacious in protecting return address from being altered, and it is a GNU C compiler extension. But StackShield has a different way to protect return address from StackGuard. StackShield saves the return address in a non-overwritten area when a function is called, and it restores the return address when the function returns. Even the return address on the stack is altered, the function can return normally. The saving and restoring process must be executed at every execution, and source code is needed to recompile the program.

Lhee and Chapin present a technique using array bounds check [14]. It is an extension to a generic C compiler. It augments executable files with type information of automatic buffers and static buffers in order to detect the actual occurrence of buffer overflow. It also maintains the size of allocated heap buffers.

Program shepherding [15] is built on top of a dynamic optimization frame work called DynamoRIO. First, it builds a custom security policy for the program using static and dynamic analysis. Then, it monitors control flow transfers during program execution to check whether the execution satisfies the security policy. Buffer overflow attack will be captured because an attack requires a control flow transfer that violates the security policy.

STOBO [16] is an instrumentation tool. It can detect buffer overflow vulnerabilities due to use of C library functions. It checks whether memory buffers satisfy certain conditions when used as parameters to library functions. Checking depends on track of the length of memory buffers.

CSSV(C String Static Verifier) [17] statically detects string manipulation errors with the aid of procedure summaries. It is up to Programmers to write procedure summaries.

CCured [18] adds memory safety guarantees to C programs by transformation. First, CCured tries to statically find possible errors in a program by enforcing a strong type system. Then, it inserts run time checks to verify whether a possible error is true.

Manish Prasad et al. present a technique of Binary Rewriting Defense [19]. It explores application of static binary translation to protect Internet software from buffer overflow attacks. A binary rewriting approach is used to augment existing Win32/Intel Portable Executable(PE) binary programs with a return address defense mechanism, which protects the return address on the stack.

CRED(C Range Error Detector) [20] builds an object tree, containing the memory range occupied by arrays, structures and unions in the program. All operations that involve pointers first locate the object in the tree to which the pointer currently refers. If locating fails, the pointer operation is considered illegal. CRED can not check accesses with library functions, and it treats structures and arrays as single memory blocks.

GMM [21] replaces certain particular functions, such as strcpy(), gets(), fscanf(), etc. The modified function saves a chunk of stack located right above the monitored function-frame and the chain composed by the previous three return addresses into a private area. When the function returns, the previous chain of return addresses is compared with the saved one. If they do not match, an exceptional handling is performed.

IBM's gcc extension [22] reorders local variables. It places pointer variables at lower addresses than the other variables. The technique offers some protection against memory pointer attacks.

PaX [23] marks data pages non-executable. It protects heap as well as stack. Because there is no execution permission bit on pages in x86 processor, PaX looks page execution as page fault. It is needed to modify the operating system.

Sagar Chaki et al. provide a technique via model checking [24]. It is static, and it is based on a paradigm called iterative refinement.

Wei Le et al. show an inter-procedural Demand-Driven Path-Sensitive analysis [25]. It classifies paths as infeasible, safe, vulnerable, overflow-user-independent, and don't-know. By using a demand-driven algorithm, the analysis is directed to those paths that can be executed and may be vulnerable.

There are many other related works [26] [27] [28].

III. MBOD(MEMORY-SIZE-ASSISTED BUFFER OVERFLOW DETECTION)

As described in section 2, there are many ways to detect buffer overflow. But there still are buffer overflow attacks, which mean more or better ways are expected. MBOD is an attempt.

A. Stack, Heap and Static Data Area

Three kinds of storage area exist in an application process space. They are stack, heap and static data area.

Stack lies in the lowest part of the process space among the three. It is allocated and released by the compiler. The compiler consecutively allocates stack, and the extension direction is from higher address to lower address.

Heap lies in the highest part of the process space among the three. It is generally allocated and released by programmer. If programmer forgets to release the allocated buffer, Operating System may release them when the application ends. The allocated buffers are not sure to be successive.

Static data area is in the middle part of the process space among the three. Static data area is used to store global variables, static variables, and constants. It is allocated when the application is compiled, and it is released when the application ends.

B. Detection Technique

The key feature of buffer overflow is that the size of the source memory is bigger than the size of the destination memory when memory copying operation occurs. In order to detect the location of buffer overflow, MBOD tries to accomplish the following three tasks:

1) Capture memory copying operation.

2) Decide the size of the source and the destination.

3) Compare the size of the source with the size of the destination.

B.1. Capture Memory Copying Operation

The data in the source buffer is not directly copied into the destination buffer when memory copying operation occurs. A register is used during copying process. First, the data in the source buffer is copied into a register, then, the data in the register is copied into the destination buffer, which means that a pair of copying operation are involved in a memory copying operation, the register(as the destination) involved in the first copying operation is the same as the register(as the source) involved in the second copying operation, and the other operands are memory buffer. Moreover, the register will not be written between the two copying operation. When such a pair of copying operation is captured, it means that a memory copying operation is captured.

B.2. Decide the Size of the Source and the Destination

The source and the destination may be a stack buffer, a heap buffer, or a static buffer. The three kinds of buffer need their respective size decision methods.

B.2.1. Decide the Size of a Stack Buffer

Stack buffer is consecutively allocated, and the extension direction is from higher address to lower address. Buffer address is the lowest address of the whole buffer.

During the execution of an application, the register esp will always point to the current position of the stack. In order to decide the size of a stack buffer, MBOD builds a stack-access chain. The stack-access chain consists of records. Each record contains such information as address-of-a-buffer and the previous record. When the register esp is modified, the address that esp points to is recorded as the address-of-a-buffer in a new record in the stack-access chain. If the address has been recorded, forget it. If the register ebp is changed, throw away the records that the relevant address-of-a-buffer is smaller than the value of ebp. Stack-access chain records some items on the stack, for example, function parameters, return address, previous frame pointer, visited local variables. The state of the stack is similar to table Ⅰ.

TAB. I.
THE STATE OF THE STACK

	Low address
Local variable n	
…	
Local variable 1	
Previous frame pointer	
Return address	
Function parameters	
	High address

When the size of a stack buffer is required, there must be a record for the buffer in the stack-access chain. The size of the stack buffer can be estimated at the distance between the buffer address and the relevant address-of-a-buffer in the nearest record. The nearest record has higher address-of-a-buffer than the buffer address.

The technique is simple and convenient.

Since the visited local variables are not sure to include all the local variables, the nearest record may record the variable next to the buffer or the variable adjacent to but not next to the buffer, or even the previous frame pointer, which means the estimated size of a stack buffer is not accurate.

In order to be more accurate, MBOD provides another technique. Stack is allocated and released by compiler. This means the size of all variables in a function and the sequence of them can be extracted from the binary code of an application. By making use of the information, MBOD builds a variable chain for every function except library functions in an application. The variable chain

consists of records. Each record contains such information as distance-of-a-buffer (d0) from frame pointer, size-of-a-buffer, and the previous record.

When the size of a stack buffer is required, first, MBOD decides to which function the buffer belongs, then, MBOD decides the distance between the buffer and the frame pointer, finally, MBOD figures out the size of the buffer.

In order to decide to which function the buffer belongs, MBOD builds a function-call chain. Function-call chain consists of records. Each record contains such information as begin-address-of-a-function, frame-pointer, the previous record, and the next record. When a function is called, a new record is inserted into the function-call chain. When a function returns, the relevant record is deleted from the function-call chain. MBOD compares the buffer address with the frame-pointer in the last record in the function-call chain. If the buffer address is bigger than the frame-pointer, MBOD Compares the buffer address with the frame-pointer in the previous record in the function-call chain. It repeats comparing and changing record until the buffer address is equal to or smaller than the frame-pointer. Then, the relevant function(f1) is the expected one, and the relevant frame-pointer is used to decide the distance(d1) between the buffer and frame-pointer.

MBOD locates the relevant variable chain according to the function(f1) found in the precious step. MBOD Compares d1 with d0 in the last record in the variable chain(d0 is the distance between the buffer recorded in the last record and the frame pointer). If d1 is smaller than d0, MBOD Compares d1 with d0 in the previous record in the variable chain. It repeats comparing and changing record until d1 is equal to d0 or d1 is bigger than d0. If d1 is equal to d0, the size in the current record is the size of the buffer. If d1 is bigger than d0, the size of the buffer is the difference between the size in the current record and (d1 - d0).

B.2.2. Decide Size of a Heap Buffer

Heap buffer(not heap area) is generally allocated and released by programmer. Programmers use memory allocation functions to allocate heap buffer, and they use memory release functions to release heap buffer. Thus, the size of a heap buffer can be obtained by tracing the memory allocation functions and memory release functions.

The allocated heap buffers are not sure to be successive. MBOD builds a heap chain to record all the allocated buffers. The record in heap chain contains such information as begin-address-of-a-buffer(a0), size-of-a-buffer, and the previous record.

When a memory allocation function is called, the expected size of the required buffer is defined as a parameter, but the address of the required buffer is unknown. Moreover, the size of the allocated buffer may be different from the expected size defined as a parameter, Therefore, the final address and size of an allocated buffer will be determined when a memory allocation function returns.

In order to obtain the needed information, MBOD traces the return instructions and makes use of the function-call chain. A new record is inserted into the heap chain when a finished memory allocation function is captured, and the relevant record is deleted from the heap chain when a finished memory release function is captured.

When the size of a heap buffer is required, MBOD compares the buffer address(a1) with the begin-address-of-a-buffer in the last record in the heap chain(a0). If a1 is smaller than a0, MBOD Compares a1 with a0 in the previous record in the heap chain. It repeats comparing and changing record until a1 is equal to a0 or a1 is bigger than a0. If a1 is equal to a0, the size in the current record is the size of the buffer. If a1 is bigger than a0, the size of the buffer is the difference between the size in the current record and (a1 - a0).

B.2.3. Decide the Size of a Static Buffer

Static data area is allocated when the application is compiled. This means the size of a buffer in the static data area can be extracted from the binary code of an application. MBOD builds a static-data chain by making use of the information. The static-data chain consists of records. Each record contains such information as address-of-a-buffer(b0), size-of-a-buffer, and the previous record.

When the size of a static buffer is required, MBOD compares the buffer address(b1) with the address-of-a-buffer in the last record in the static-data chain(b0). If b1 is smaller than b0, MBOD Compares b1 with b0 in the previous record in the static-data chain. It repeats comparing and changing record until b1 is equal to b0 or b1 is bigger than b0. If b1 is equal to b0, the size in the current record is the size of the buffer. If b1 is bigger than b0, the size of the buffer is the difference between the size in the current record and (b1 - b0).

B.3. Compare the size of the source with the size of the destination

Comparing is simple after memory copying operation is captured and the size of the source and the size of the destination are figured out. Latent buffer overflow is reported if the size of the source buffer is bigger than the size of the destination buffer.

IV. IMPLEMENTATION

MBOD is implemented as a Pintool. The static information extracted from binary code is not obtained directly. It obtains static information through the corresponding assembler code of the binary code of the analyzed application. The assembler code is generated by IDA Pro. The tested applications are only C applications now, and it runs on the windows platform, therefore, the format of tested applications is PE(Portable Executable).

A. Software Architecture

Pin is a software instrumentation tool. It provides easy-to-use, portable, transparent, and efficient instrumentation. Instrumentation tools(called Pintool) are written in C/C++ using Pin's API. Pin allows the tool writer to analyze an application at the instruction level. It uses dynamic compilation to instrument executables while they are running.

IDA Pro is a programmable, interactive disassemble and debugger. It is one of the most popular disassembling tools for Windows. Assemble code can be shown in the user interface of IDA Pro, and it can be written into a file to utilize for other tasks.

Pin is a dynamic instrumentation tool. This means only one path of an application is executed in one execution. In order to detect all possible buffer overflows, all paths should be executed. MBOD starts Pin one time for every path of an application. In order to achieve the goal, MBOD controls selection(fall-through or branch-target) when a branch shows, and saves all branches and the relevant selections when an execution ends, and restores them when the next execution begins.

MBOD consists of two parts, namely initialization, and capturing.

B. Initialization

As described in section 3, the methods to determine the size of different types of buffer are different. This means the type of a buffer must be determined before the size of a buffer is determined. As presented in section 3, different types of buffer lie in different parts of the process space of an application. According to the address of a buffer, MBOD decides the buffer is a stack buffer if the address is smaller than the begin address of the data segment, a heap buffer if the address is bigger than the end address of the data segment, and a static buffer if the address is bigger than the begin address of the data segment but smaller than the end address of the data segment. For the sake of determining, MBOD obtains the begin address and the end address of the data segment at the initialization stage.

Since the size of a heap buffer is needed, memory allocation functions and memory release functions must be traced. In order to identify the memory allocation functions and memory release functions while an application is running, MBOD must know the name of these functions and the relevant begin address of these functions. The name information does not exist while an application is running in Pin. IDA Pro provides the name and the begin address of these functions. MBOD obtains them by making use of IDA Pro before an application is started in Pin.

MBOD builds a branch tree to save the branch information of an application. Branch tree consists of branch records. Branch record contains such information as address-of-branch-instruction, father-branch, left-son-branch, right-son-branch, address-of-father-branch, address-of-left-son-branch, address-of-right-son-branch, left-son-executed-or-not, right-son-executed-or-not. Branch tree is built at the initialization stage.

At the initialization stage, MBOD also builds stack chain and static-data chain by utilizing IDA Pro. Stack chain holds a variable chain for every function, and as described in section 3, static-data chain holds a record for every static buffer.

C Capture

MBOD inserts instrumentation functions before or after certain particular instructions to accomplish capturing.

For all the instructions that write memory, MBOD inserts instrumentation function before them to capture address of the memory that will be written.

For all the instructions that read memory, MBOD inserts instrumentation function before them to capture address of the memory that will be read.

Registers are very important during execution of an application. They hold much information about execution process. Tracing their values is a good way to know the execution process in detail. For all the instructions that modify registers, MBOD inserts instrumentation function after them to save the value of registers esp, ebp, esi, edi, ecx, edx, ebx, and eax, and to update stack-access chain when the value of register esp is modified.

Buffer overflow happens when memory copying operation occurs. For all the instructions that move data between memory and register, MBOD inserts instrumentation function after them to accomplish the following tasks:

1) Find move instruction(ins1) that reads data from memory(m1) and write data to register(r1).

2) Save (name(n1) of register r1, address(a1) of memory m1, size(s1) of memory m1) of instruction ins1.

3) Find move instruction(ins2) that reads data from register(r2) and write data to memory(m2).

4) Save (name(n2) of register r2, address(a2) of memory m2, size(s2) of memory m2) of instruction ins2.

5) Compare n2 with n1. Ins1 and ins2 are considered to perform a memory copying operation if n2 is the same as n1 and r1 is not written between ins1 and ins2.

6) Compare s2 with s1. Buffer overflow is reported if s2 is smaller than s1.

In order to make each path of an application executed once and only once, it is needed to control the selection of branch instruction. For all the branch instructions except direct jump instructions, MBOD inserts instrumentation function before them to achieve jumping directly to the next instruction of the branch instruction or to the target instruction of the branch instruction.

For all call instructions, MBOD inserts instrumentation function before them to accomplish the following tasks:

1) Insert record to function-call chain.

2) Save address and target address of the call instruction.

For all return instructions, MBOD inserts instrumentation function before them to accomplish the following tasks:

1) Save the size of the allocated buffer and frame pointer for return instructions of memory allocation functions.

2) Insert record to heap chain for return instructions of memory allocation functions.

3) Delete record from heap chain for return instructions of memory release functions.

4) Delete record from function-call chain.

When execution of application ends, MBOD inspects branch tree to decide whether two selections of all branches have been executed. If not, saves branch tree in a file, otherwise, ends the detecting process.

V. Results and Discussion

MBOD is employed to detect buffer overflow for some applications. The following one is an example.

```
#include "stdio.h"
char dst[20];
void vulFunc(char *s) {
   char buf[20];
   strcpy(buf, s);
   printf("vulFunc String=%s\n", buf);
}
void vulFunc2(char *s) {
   char buf[20];
   sprintf(buf,"%s",s);
   printf("vulFunc2 String=%s\n", buf);
}
main(int argc, char *argv[]) {
   FILE *buffer_overflow_input;
   if(argc == 3) {
      if( atoi(argv[1]) )
         vulFunc(argv[2]);
      else {
         vulFunc2(argv[2]);
      }
   }
   else if(argc == 2) {
      if( atoi(argv[1]) )
         vulFunc(argv[1]);
      else {
         buffer_overflow_input=fopen("C:/download/15/Pin/src_application/buffer_overflow.input","r");
         if(buffer_overflow_input!=NULL) {
            fgets(dst,30,buffer_overflow_input);
            fclose(buffer_overflow_input);
            printf("1 no call of vulFunc.\n");
         }
         else
            printf("2 no call of vulFunc.\n");
      }
   }
   else {
      printf("Usage: %s ,call vulFunc or not(1/0) ,input string\n",argv[0]);
   }
}
```

It is obvious that there are four buffer overflows in it. MBOD provides the following report after inspecting the relevant binary code of the application.

```
1)
0x774ea9bb(0x75e53821)->0x75e53831(0x401428:start)
   ->0x4014d7(0x40105b:_main)->0x401081(0x401000:sub_401000)
   ->0x40100e(0x401170:_strcpy)
2)
0x774ea9bb(0x75e53821)->0x75e53831(0x401428:start)
   ->0x4014d7(0x40105b:_main)->0x401092(0x40102b:sub_40102B)
   ->0x40103e(0x401260:_sprintf)->0x401289(0x40161a:__output)
   ->0x401d02(0x401dc1:_write_string)->0x401de1(0x401d5b:_write_char)
3)
0x774ea9bb(0x75e53821)->0x75e53831(0x401428:start)
   ->0x4014d7(0x40105b:_main)->0x4010bf(0x401000:sub_401000)
   ->0x40100e(0x401170:_strcpy)
4)
0x774ea9bb(0x75e53821)->0x75e53831(0x401428:start)
   ->0x4014d7(0x40105b:_main)->0x4010ee(0x401308:_fgets)
```

The report shows that there are four buffer overflows. They are respectively in strcpy() in sub_401000(), sprintf() in sub_40102B(), strcpy() in sub_401000() (has different call position from the first one), and fgets() in main(). The report is satisfying. But MBOD is not always so good.

One defect of MBOD is that it has not been employed to large application successfully now, because it is not easy to debug execution based on instruction level instrumentation. Another defect of MBOD is that it will break when the inspected application breaks.

VI. Conclusions

MBOD is an attempt to detect buffer overflow. It possesses the following advantages:

1) Binary code instead of source code of an application is needed.

2) It tries to detect the possible buffer overflow rather than prevent a triggered off buffer overflow from advancing, therefore, it can find vulnerabilities in programs even when the tested data does not trigger off an overflow.

3) The detection process needs to be executed only once, not to be executed whenever the application is executed, so the overhead is low.

4) It detects all forms of buffer overflow.

But, as described in section 5, two defects exist in it. Improvements should be taken to make MBOD more practical.

References

[1] C. Cowan, P. Wagle, C. Pu, et al. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In Proceedings DARPA Information Survivability Conference and Exposition, pages 119-129, Hilton Head, SC, Jan. 2000.

[2] Alephone. Smashing the Stack for Fun and Profit. Phrack, 7(49), Nov. 1996.

[3] Klog. The Frame Pointer Overwrite. Phrack Magazine 55(8), May 2000.
[4] Chi-Keung Luk, Robert Cohn, Robert Muth, etc. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. Proceedings of the 2005 ACM SIGPLAN conference on programming language design and Implementation, Chicago IL USA, pages 190-200, 2005.
[5] T. M. Austin, S. E. Breach, G. S. Sohi. Efficient Detection of all Pointer and Array Access Errors. In ACM SIGPLAN 94 Conference on Programming Language Design and Implementation, June 1994.
[6] R. W. M. Jones, P. H. J. Kelly. Backwards-compatible Bounds Checking for Arrays and Pointers in C Programs. In Proceedings of the third International Workshop on Automatic Debugging, pages 13-26, Sweden, May 1997.
[7] Solar Designer. Non-executable Stack Patch. http://www.openwall.com/linux December 1997.
[8] C. Cowan, C. Pu, D. Maier, et al. Stackguard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In Proceedings of the 7th USENIX Security Symposium, pages 63-77, San Antonio, TX, Jan. 1998. USENIX.
[9] Baratloo A, Singh N, Tsai T. Libsafe: Protecting Critical Elements of Stacks. http://www.research.avayalabs.com/project/libsafe, 1999.
[10] A. Baratloo, N.Singh, T. Tsai. Transparent Runtime Defense Against Stack Smashing Attacks. In Proceedings of the 2000 USENIX Annual Technical Conference, pages 251-262, San Jose, CA, June 2000. USENIX.
[11] D. Wagner, J. S. Foster, E. A. Brewer, et al. A first Step towards Automated Detection of Buffer Overrun Vulnerabilities. In Network and Distributed System Security Symposium, pages 3-17, San Diego, CA, Feb. 2000.
[12] D. Larochelle, D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In Proceedings of the 10th USENIX Security Symposium, Washington D. C, Aug. 2001. USENIX.
[13] Vendicator. StackShield. http://www.angelfire.com/sk/stackshield. 2001.
[14] Kyung-suk Lhee, Steve J. Chapin. Type-Assisted Dynamic Buffer Overflow Detection. In Proceeding of the USENIX Security Symposium, pages 81-89, August 2002.
[15] Vladimir Kiriansky, Derek Bruening, Saman Amarasinghe. Secure Execution Via Program Shepherding. 11th USENIX Security Symposium, August 2002, San Francisco, California.
[16] E. Haugh, M. Bishop. Testing C Programs for Buffer Overflow Vulnerabilities. In Proceedings of the Network and Distributed System Security Symposium, February 2003.
[17] N. Dor, M. Rodeh, M. Sagiv. Cssv: Towards a Realistic Tool for Statically Detecting all Buffer Overflows in C. In Proceedings of ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pages 155-167, June 2003.
[18] J. Condit, M. Harren, S. McPeak, et al. CCured in the Real World. In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, June 2003.
[19] Manish Prasad, Tzi-cker Chiueh. A Binary Rewriting Defense against Stack based Buffer Overflow Attacks. Proceedings of USENIX'03 Annual Technical Conference, San Antonio, Texas, USENIX Press, 2003, 211-224.
[20] Olatunji Ruwase, Monica S.Lam. A Practical Dynamic Buffer Overflow Detector. Proceedings of the Network and Distributed System Buffer Overflow Symposium, pages 159-169, February 2004.
[21] Davide Libenzi. Guarded Memory Move Buffer Overflow Detection and Analysis. http://www.xmailserver.org/gmm.pdf, 2004.
[22] Hiroaki Etoh. GCC Extension for Protecting Applications from Stack-smashing Attacks. http://www.trl.ibm.co.jp/projects/security/ssp, August 2005.
[23] PaX. http://pageexec.virtualave.net October 2005.
[24] Sagar Chaki, Scott Hissam. Precise Buffer Overflow Detection via Model Checking. http://www.sei.cmu.edu/staff/chaki/publications/WhitePaper-2005-2.html, 2005.
[25] Wei Le, Mary Lou Soffa. Refining Buffer Overflow Detection via Demand-Driven Path-Sensitive Analysis. Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, security and fault detection, pages 272-282, 2008.
[26] Yang Xu, Xiaoyao Xie. Modeling and Analysis of Security Protocols Using Colored Petri Nets. Journal of Computers 6(1), January 2011, 11-18.
[27] Bo Meng, Wei Wang, Wei Chen. Verification of Resistance of Denial of Service Attacks in Extended Applied Pi Calculus with ProVerif. Journal of Computers 7(4), April 2012, 890-899.
[28] Bo Meng, Wei Huang, Zimao Li. Automated Proof of Resistance of Denial of Service Attacks Using Event with Theorem Prover. Journal of Computers 8(7), July 2013, 1728-1741.

Chunguang Kuang's research projects center on software vulnerability analysis. She graduated from Changsha Institute of Technology in 1993 with a B.S. in computer. In 2008, she earned a M.S. in computer science from Beijing Institute of System Engineering. She was promoted to associate research fellow in 2003.

Chunlei Wang's research projects center on software security. He graduated from Institute of Command and Technology of Equipment in 1999 with a B.S. in computer. In 2002, he earned a M.S. in computer science from Institute of Command and Technology of Equipment. He is studying in Tsinghua University for a Ph.D. in computer security.

Minhuan Huang's research projects center on network security. He graduated from Changsha Institute of Technology in 1993 with a B.S. in electronic technology. In 2002, he earned a M.S. in computer science from Changsha Institute of Technology. He is studying in Tsinghua University for a Ph.D. in computer security. He was promoted to associate research fellow in 2002, and he was promoted to research fellow in 2009.