ISSN (Print) : 2319-5940
ISSN (Online) : 2278-1021

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 2, Issue 5, May 2013

A Categorized Survey on Buffer Overflow Countermeasures

Jisha S[1], Diya Thomas[2], Sangeetha Jamal[3]

M. Tech Scholar, Department of Computer Science, Rajagiri School of Engineering and Technology, Kochi, India [1]

Asst. Professor, Department of Computer Science, Rajagiri School of Engineering and Technology, Kochi, India [2,3]

Abstract: Buffer overflow vulnerability is a fundamental cause for most of the cyber attacks such as server breaking-in, worms, zombies, and botnets, since the attacker gets a capital control over a victim host. Many solutions to the buffer overflow attacks have been proposed in the last decade. However, on a routine basis new buffer overflow vulnerabilities are still discovered and reported. Since almost all existing solutions to the buffer overflow attack problem require significant modification to the computing infrastructure in which network applications are developed or executed, and thus have met considerable resistance in actual deployment. This paper is aimed to provide a categorized survey for the existing countermeasures to buffer overflow attack. A categorized survey is necessary in this field because researchers have proposed many software-based and hardware based countermeasures for buffer overflow exploits. These methods differ from one another in the strength of protection provided, the effects on performance, and the easiness of deployment. Finally, the paper compares the effectiveness, performance and limitations of the different category.

Keywords: Buffer overflow attack, Cyber security, Operating System, Computer Architecture

I. Introduction

The first well known exploit of buffer overflow vulnerability is occurred in 1988 when the infamous Internet Worn shutdown over 6,000 systems in just a few short hours, exploiting gets() function call in the *fingerd* daemon process [1]. Today also, the buffer overflow continues to be a significant and prominent computer security concern. Monitoring program helps computer systems from malicious code injection attacks and to recover from soft errors. There has been a lot of work on this area. However, most work targets on individual problem only. Another solution used for preventing buffer overflow attack is by compiler extension. This include checking compiled binary for known vulnerable functions, performs some data and control flow analysis, and checking for correct boundaries. Operating system based solutions declare stack as non executable and hence prevent code execution. In hardware based solutions illegal instructions and modifications are inspected. Defense side obfuscation techniques allow some obfuscation to be made on the host machine hence the attacker will be in great trouble to obtain information specific to that machine. Another method of prevention is to capture code running symptoms and prevent code injection attack. This involves identifying anomalous sequences of system calls executed by programs.

The paper is organized as follows. Section II describes buffer overflow attack. Categorization of the countermeasures against buffer overflow attack is done in section III. In section IV, we are performing a comparison of the discussed methods in section III and section V concludes the paper.

II. Buffer Overflow attack

A buffer overflow occurs during program execution when too much data copied into a fixed-size buffer. This causes the data to overwrite the adjacent memory locations. Depending on what is stored there, the behavior of the program is changed. That results erroneous program behavior, system crash, memory access errors etc. A buffer is contiguous allocated memory. When the program is in execution, the memory allotted for the program contain a set of binary instructions to be executed by the processor; some read-only data; global and static data for the program whose scope is throughout the program execution.

A Linux process memory layout is shown in fig.1. The memory layout begins with program code and data. It contains program instructions and initialized and uninitialized static and global data; followed by a run time heap. The run time heap is created by malloc/calloc; which is followed by the user's stack. When a function is called the stack is used. A stack is a contiguous block of memory. Whenever a function is called, its parameters return address, and fame pointer (FP) is pushed in order onto the stack. Stack grows from higher memory addresses to the lower ones.

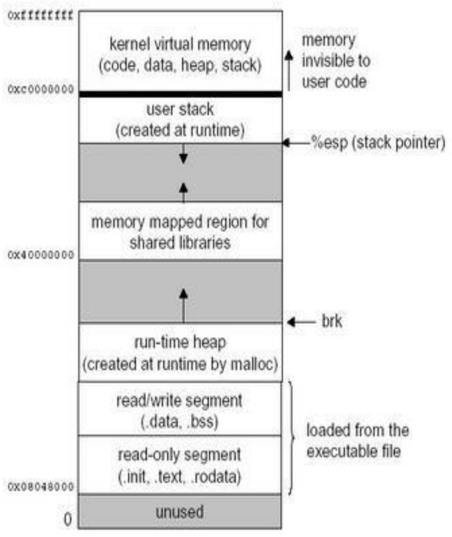
0xffffffff
kernel virtual memory (code, data, heap, stack)
memory invisible to user code
0xc0000000
user stack (created at runtime)
%esp (stack pointer)
memory mapped region for shared libraries
0x40000000
brk
run-time heap (created at runtime by malloc)
read/write segment (.data, .bss)
loaded from the executable file
read-only segment (.init, .text, .rodata)
0x08048000
0
unused

Fig. 1. Memory layout of Linux program

Consider the following C program.

```
void foo (char *str) {
  char buf[15];
  strcpy (buf, str);
}
int main () {
char *str = "I am greater than 15 bytes"; // length of str = 27
//bytes
 foo (str);
}
```

This program shows unexpected behavior, because a string (str) of 27 bytes has been copied to buffer that has been allocated for only 15 bytes. The extra bytes overwrite the space allocated for the FP, return address and so on. The data used to overflow is often a string crafted by the attacker which contains executable code and repetitions of the target address which over write the return address. After successfully modifying the return address the attacker is able to execute instructions with the same privileges as that of the attacked program.

III. CATEGORIZED COUNTERMEASURES

All paragraphs must be indented. All paragraphs must be justified, i.e. both left-justified and right-justified.

A. Finding Bugs in the Source Code

In C program, there are lot of vulnerable library functions which include: *gets(), strcpy(), strcat(), sprintf(), vsprintf(), fscanf(), sscanf(), vscanf(), vsscanf(), vfscanf(), realpath(), getopt(), getpass(), streadd(), strecpy(), strtrns(), syslog()*.The *scanf ()* family of functions may also result in buffer overflows. One way to try to avoid buffer overflow vulnerabilities from software is to inspect the source code and look for the vulnerabilities. This can be done by manually looking at all the source files line by line, or by using the UNIX *grep* command to look for vulnerable library functions, such as *strcpy* or *gets*. Some of the vulnerabilities are caught by this technique, but by no means completely guarantees the safety of the resulting code. Manual review will miss many buffer overflow vulnerabilities due to the complex interactions of the software (eg. Microsoft Windows). A certain code section may look safe, but when one takes into account the different interactions, it could be completely unsafe. Also, *grep* may be able to find all instances of *strcpy* in the source code, but it still requires a human to interpret its usage in the code to tell if it can lead to a buffer overflow. In addition, functions such as *strcpy* are not the only source of buffer overflow vulnerabilities. Programmers, even careful ones, can introduce new vulnerabilities with any kind of algorithm that uses buffers.

So while programming with C, take special care when using the vulnerable function and it is best to use *strncpy(), strncat(), snprintf(), vsnprintf()* instead of vulnerable functions. Some solutions transform static buffers to dynamically allocated heap-based buffers so that any overflow to these buffers leads to a segmentation fault and thus an exploit attempt can be find out. In Reference [2], a code segment is generated to detect out of bound accesses and when out of bound access occurs, instead of letting it to corrupt the memory it is stored in a hash table, and whenever the value is referenced they will provide the stored value based on read address and allow the program to continue execution instead of crashing/halting.

In [3], a tool based on LCint is used. In this paper they propose some new annotations- “ensures” and “requires” through which programmers can state function’s pre-conditions and post-conditions. And they are using several constraints such as “minSet”, “maxSet”, “minRead” and “maxRead”. These constraints describe the range of buffers used in the program. When a function is called pre-conditions and post-conditions are verified to ensure safe access of buffers using the buffer range constraints. This method requires the programmers to provide annotations and protects such annotated functions.

All the above methods provide a counter measure for buffer overflow attack by looking the source code for vulnerability.

B. Compiler Extensions

A buffer overflow prevention method proposed by [4] is StackGuard. This is a compiler extension which places a “canary” between local variables and the return address on the stack. This canary is randomly generated when the

ISSN (Print) : 2319-5940
ISSN (Online) : 2278-1021

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 2, Issue 5, May 2013

program begins to run. It is a 4 byte number. When a function completes, before the control is transferred to the return address which was on the stack, the canary is checked with its original value. If the value does not match, then it can be concluded that an attacker overflowed a buffer in the function that just completed and the program will terminate. This effectively detects when a buffer overflow occurs and kills the process before it has a chance of executing the attack code.

Tzi-cker Chiueh et..al.(2001) are discussing a method called RAD[5]. *RAD* is a patch to gcc-2.95.2 that automatically adds protection code into the function prologues and epilogues of the programs compiled by it. So the source code does not need to be modified. This is a compiler extension technique. By overflowing a return address with a pointer to the injected code, attackers can have the code executed with the attacked program's privilege. Return address defender (RAD) prevents this by making a copy of the function return address in a particular area of the data segment called *Return Address Repository* (RAR). By setting neighboring regions around RAR as read-only, we can defend RAR against any attempt to modify it through overflowing. Given that RAR's integrity is guaranteed, each time when a return address of a stack frame is used to jump back to the caller function, this address is checked with the copy in RAR. A return address will be treated as un-tampered and thus safe to use only if RAR also contains the same address. In the paper they are proposing two versions of RAD, MineZone RAD and Read-Only RAD. Both these methods protect the return addresses stored in RAR in two different ways. In MineZone RAD, they are creating a C file, /hacker/global.c which is automatically linked with programs compiled by RAD. This C file contains all the function definitions and variable declarations used in the new function prologues and epilogues. In global.c they declare a global integer array which is divide it into 3 parts as shown in fig. 2.

The middle part of the global integer array is RAR, which keeps a redundant copy of the return address of each function call. The first and third parts, call *mine zones*, are set as read-only areas by *mprotect()* system call. All protection functionalities are implemented as instructions added to the new function prologues and epilogues without changing the stack frame layout of each function. Therefore programs compiled by RAD are compatible with existing libraries and other object files. In the new function prologue, the first instruction executed is "pushing a copy of the current return address into RAR." Any attempt to overwrite the RAR would cause a trap and is denied by the OS.

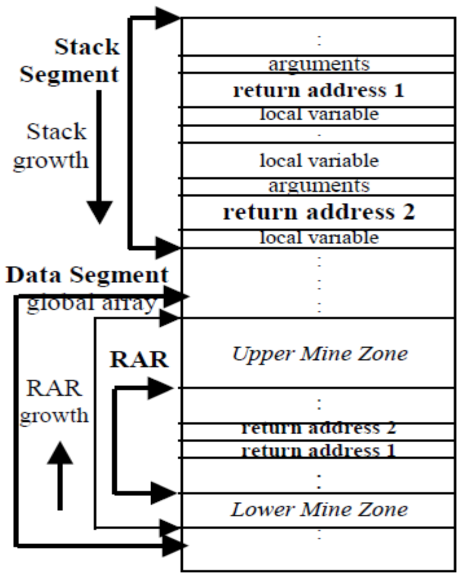

Fig. 2. Structure of RAR and MineZones

Read-Only RAD is similar to Mine-Zone RAD. It sets the RAR itself as read-only to protect itself. The only time that it becomes writable is in the function prologues when the current return address is pushed into RAR. No external input statements are there in the function prologue. To update RAR in function prologue requires adding two extra system calls to each function call, causing a serious performance penalty.

Point Guard [6] is another compiler technique which protects attacks against pointers. While the program is in memory, the pointers are encrypted using pre-process XOR key and when they are loaded into registers, they are decrypted.

There are many techniques such as StackShield [7] which have resemblance to the above discussed methods. All those methods also include compiler modification for their counter measure, so they can be included in this category.

C. Hardware Modifications

SmashGaurd [8] aimed to protect return address, is a hardware solution against buffer overflow attacks. Here the return address is stored in a hardware stack added to the CPU. For each function call instruction the return address and the stack frame pointer is pushed onto the hardware stack. A return instruction pops the most recent pair of address from the top of the hardware stack and compares it with the return address. If any mismatch occurs, a hardware exception is raised. In this approach, all reads and writes to the hardware stack are done in hardware, through the function call and return instructions. No other instruction is permitted to read/write directly from or to the hardware

stack. Hence this method do not involve any changes to the application code.
Reference [9], also discuss about a processor architecture defense against buffer overflow attack. They also use a secure return address stack, to provide a built-in dynamic protection against return address corruption. This method does not require any change to the application code.

D. Operating System based Solutions

A robust kernel based solution called *AURORA* [10] is proposed to prevent control hijacking buffer overflow attack. Control hijacking means overwriting control sensitive data such as return address, function pointers and *jumpbuf* with new address. The defense method includes blocking attack traffic in the operating system kernel before attack traffic destroys the address space of an attacked process. Then it sends a socket close or end-of –file to the attacked process in response to the service request and close the related socket/file. The major component of *AURORA* is *Memory Area Observation Method* (MAOM). Through this method *AURORA* can understand whether the *read/recv* system call inside kernel directly overwrite the return address or the caller *ebp* field of the stack frame of any active function of the process. For this *MAOM* use address of the input buffer and length of the input string. However *MAOM* could not detect indirect overflow [overflow occur to input buffer is called direct overflow, otherwise it is called indirect overflow] and direct overflow that overwrite a function pointer. To detect these overflow signatures are used.

For signature creation, *AURORA* use indispensable properties and important properties of attack payload elements. Indispensable property is the property that is created to a successful attack. Important property is the property that increases the chance of a successful attack. Since signature is based on indispensable property and important property, *AURORA* can detect zero-day control hijacking buffer overflow attack.

E. Defence Side Obfuscation

Reference [11] describe a randomized instruction set emulator (RISE) based on the open source Valgrind X86-to-X86 binary translator. This technique obscures the machine instruction set using a private randomized scrambling mechanism. In order to put a successful binary code attack, the attacker should obtain information specific to the machine which is very hard to attain. The scrambling function is designed in such a way that it is very hard to create code sequences to perform a desired function (e.g.: an attack), since it need a long secret key which is unique to each program execution. Hence when binary attack code comes to the system, it will appear as a random string of bits. Source languages that are vulnerable to programming error, local and trusted programs and machine runs mostly local (trusted) programs are equally co-operative to randomization strategy.

The encryption scheme includes selecting a key of length n bytes, where n is a parameter of the system. XOR the key with the first n bytes of the machine code, and till the executable is scrambled the operation is repeated. The key is generated randomly for every new process. When decoding, the byte to be decoded is XOR-ed with its corresponding part (subkey) of the key. The subkey index in the key is easily recovered from the instruction pointer (EIP) by the operation, EIP (mod n). This allows memory that was encoded linearly to be decoded correctly regardless of the order of instruction execution, even if x86 instructions have varying lengths.

F. Capturing Code Running Symptoms

This type of counter measure works by detecting whether malicious program running on the system or not [12]. Based on this, Ref. [13] proposed an effective protection against large scale attacks. This approach consists of four phases such as attack detection, input correlation, attack localization and signature generation.
For the first phase- *attack detection*, a memory error exploit protection technique-Address Space Randomization (ASR) is used. *ASR* randomizes the location of various objects (executable code, shared libraries, stack, heap, and static data) in process address space. Thus, even though an attacker can control the value of a pointer, he cannot confirm that the pointer references a valid memory location being used by the program. Most programs use only a small fraction of the address space available to them; hence probability of choosing a valid location is very small. So dereferencing the pointer will lead to a memory exception with a high probability. An attack is detected for such memory exception which raises a segmentation fault, bus error or illegal instruction signal and triggers the next phase.

The second phase-*input correlation* uses a signal handler. The signal handler can query the operating system to identify the memory address that caused the exception, and thus identify the value used to overwrite the pointer. Then the system will identify the recent input string that contains the value. If multiple matches are found, all of them are marked as candidates for next step.

After identifying the input that involved in the attack, in the third phase-*Attack localization in input*- the system map this input to a particular message type and/or field and the signature will be based on this field.

For signature generation a simple, light-weight rule generation algorithm is used that exploits unique features of buffer overflows. It considers all available benign input samples to ensure none of these inputs will be filtered out. The system also maintains certain summary statistics about all benign inputs, which can be quickly compared with those of malicious inputs when they are encountered.

A similar approach named *ARBOR* is proposed in [14] where authors are tried develop a self healing system to protect network server processes. An *input filter* is used in this system which reject input that match with existing filter rules. The rules generated by the *analyzer* are aimed to capture characteristics of attack bearing input. Another component is a *behavioral model* which is consulted by all the components of *ARBOR*. The behavioral model is constructed from the system calls and other relevant library calls which are intercepted by another component called the *logger*. The behavioral model provides the basis for filter generation logic in the *analyzer.* It also provide services to the *input filter* to carry out input tests. The *logger* is focusing the library calls, since Zhenkai Liang et. al argues that the library call interposition will cause only low overhead when compared to system call interposition. The *logger* will record sampled (to reduce overhead) subset of input events used by the *analyzer.*

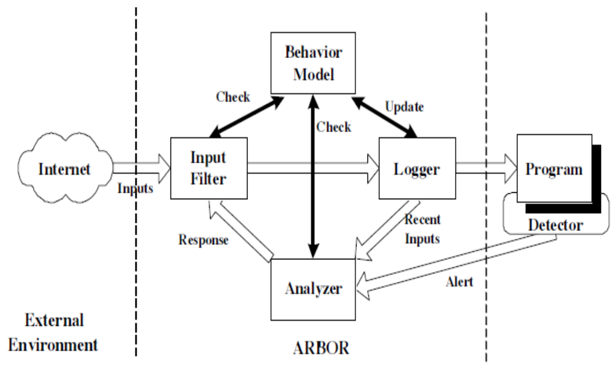

Fig. 3. Architecture of ARBOR

The *detector* uses address obfuscation to detect buffer overflow attempt. When an attack occurs on the server crashes which will trigger the feedback loop in *ARBOR*. If the protected process is attacked, a notification is send to the *detector*. Then the *analyzer* collects recent program behavior from the *logger*. It also examines the recent inputs to the attacked process. The *analyzer* is responsible for synthesizing filter rules. For that it defines a threshold depending on the size of input when the process was running normally and allowable maximum size for the input. With this concept *analyzer* synthesize a filter rule. This filter rule will flag an attack if the input size is greater than the threshold.

ARBOR also considers the program's execution context of input operation to identify attack. The relevant context information include execution path taken by the program, the content of runtime stack at the time of input operation, parameters to the input operation etc. In this way *ARBOR* prevent compromising with buffer overflow attack by capturing the input behavior and program behavior.

VI.DISCUSSIONS

Inspecting the source code for bugs will helps to find out program errors before the compilation. But this approach has several disadvantages. This method need source code and for many legacy applications, source code is not available. Only small projects can be protected by this method.

RAD and StackGuard are considered approximately based on same strategy. RAD uses RAR and StackGuard uses canary words to prevent injected addresses from being used as return addresses of function calls. The run time address space of a program compiled by RAD contains two copies of return addresses, one in the stack and one in RAR. Any attempt to change return addresses in the stack will be detected by RAD and result in the termination of the program and the delivery of a warning message to root. RAR is protected by system call mprotect().

In StackGuard, if the hacker can correctly guess the canary value, the protection is not guaranteed. Here the solution is to use random canary instead of a static canary value. The attacker is also able to skip over the canary word and overwrite the return address by using alignment requirement. But MineZone RAD gives more protection here, because the attacker must change both the return address in stack and RAR.

StackGuard, ProPolice[15], Stack Shield [7], and RAD are based on checking the integrity of return address before a return, and hence the attack will be detected just before the corrupted pointer is used by the program. With StackGhost, PointGuard and ASR, detection occurs right after the corrupted pointer is used.

AURORA does not need modification of the source code of any application programs and also compatible with existing operating systems and application programs; hence, AURORA could work with other protection mechanisms to provide an extra layer of protection. When an attack occurs, the attacked process becomes idle because the processes continue waiting for attackers' input which has already been blocked by a protection mechanism. A process crash occurs due to the destruction of address space of an attacked process. AURORA guarantees elimination of process idleness and repeated process crashes.

Randomized instruction set emulation technique also disrupts binary code injection attack without program recompilation, linking. And also it does not need access to program source code.

PointGuard rely on encrypting the vulnerable pointer with a random XOR mask. As a result, when the overwritten pointer is referenced, it leads to dereference of a random location in memory, with a very high probability of causing a memory fault.

ISSN (Print) : 2319-5940
ISSN (Online) : 2278-1021

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 2, Issue 5, May 2013

TABLE I
COMPARISON OF DIFFERENT COUNTERMEASURES

Methods	Source Code	OS Modification	Compiler Extension	Hardware Modification	Run time Monitoring	Obfuscation
StackGuard	No	No	Yes	No	No	No
RAD	No	No	Yes	No	No	No
PointGuard	No	No	Yes	No	No	No
SmashGuard	No	No	No	Yes	No	No
Randomized Instruction Set Emulator	No	No	No	No	No	Yes
ARBOR	No	No	No	No	Yes	No
AURORA	No	Yes	No	No	No	No

In ARBOR, generation of signature based on the complete input may lead to increased false alarms. If signature generation is based on the field, it will reduce the likelihood of false matches with signatures; thereby minimize the possibility of rejection of legitimate request. Also attacks may be delivered through a sequence of small packets, causing each input operation to return a small amount of data. Thus matching operation will fail. And if the buffer overflow attack is triggered by a field in the request, and not the entire request. Then, the length of input is not a characteristic of the attack. The comparison Table I. shows the necessary changes that are needed for a counter measure method.

IV. CONCLUSION

Buffer overflow attack will cause very serious problem, unless we take effective countermeasure. In this paper we have discussed different category of prevention method. Each category has its own advantage and disadvantage. If we are using category A, the source code should be available at the defense side and for many legacy applications, source code is not available. For category B, existing compiler for applications should be modified. But this technique does not need source code for their attack defense. Similar issues arise with category C and category D. With category E, care should be taken to minimize the process restart cost. Category F suffers from significant run time overhead. But it is more secure and economical if runtime overhead can be reduced to an acceptable range. The majority of buffer overflow attacks involve overwriting procedure return address in the memory stack. So the best way to prevent Buffer overflow attack is to enforce protection at the lower level like instruction set randomization which will scrambles the binary code at load time and unscrambles instruction-by-instruction during instruction fetch and execute the unscrambled code correctly. But this technique also got some disadvantage. That is, when scrambling the bit randomly, it will produce another legal instruction which is executable. Hence with the already proposed methods there are many issues. Due to severity of Buffer overflow attack, the proposals of new solution which are simple to maintain; transparent to existing hardware, Operating system and application software; and economical are very important necessity in the field of cyber security.

REFERENCES

[1] SANS Institute InfoSec Reading Room, http://www.sans.org
[2] Rinard M., *A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities and Other Memory Error*, in 20th Annual Computer Security Applications Conference. 2004, IEEE Computer Security, pp. 82-90.
[3] Ganapathy V, *Buffer Overrun Detection using Linear Programming and Static Analysis*, 10th ACM conference on Computer and communications security 2003, ACM: Washington D.C., USA. pp. 345-354.
[4] C.Cowan, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, *Stackguard: Automatic Dectection and Prevention of Buffer-overflow Attacks,* In Proceedings of the 7th USENIX Security Symposium, Jan. 1998.
[5] Tzi-cker Chiueh, Fu-Hau Hsu, *RAD: A Compile-Time Solution to Buffer Overflow Attacks*, Proc. 21st Int'l Conf. Distributed Computing Systems (ICDCS '01), pp. 409-417, Apr. 2001.
[6] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, *Pointguard: Protecting Pointers from Buffer Overflow Vulnerabilities*, Proc. 12th USENIX Security Symp., pp. 91-104, Aug. 2003.
[7] Vendicator, *StackShield: A 'Stack Smashing' Technique Protection Tool for Linux*, http://www.angelfire.com/sk/stackshield/ , Jan. 2001.

ISSN (Print) : 2319-5940
ISSN (Online) : 2278-1021

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 2, Issue 5, May 2013

[8] Hilmi O. zdoganoglu, T.N. Vijaykumar, Carla E. Brodley, Benjamin A. Kuperman, and Ankit Jalote, *SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address*, IEEE Transactions On Computers, Vol. 55, No. 10, Oct. 2006.

[9] J.P.McGregor, D. K. Karig, Z. Shi, and R. B. Lee, *A Processor Architecture Ddefense against Buffer Overflow Attacks*, In Proc. Int. Conf. Inf. Technology Res. Edu., 2003, pp. 243–250.

[10] Li-Han Chen, Fu-Hau Hsu, Cheng-Hsien Huang, Chih-Wen Ou, Chia-Jun Lin And Szu-Chi Liu, *A Robust Kernel based Solution to Control-Hijacking Buffer Overflow Attacks*, Journal of Information Science and Engineering, Vol. 27 No. 3, pp. 869 -890 May 2011.

[11] Barrantes, E. Ackley, D. Palmer, T. Stefanovic, D. Zovi, *Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks*, In Proceedings of the 10th ACM conference on Computer and communications security Oct. 2003.

[12] J. Rabek, R. Khazan, S. Lewandowski, R. Cunningham, *Detection of Injected, Dynamically Generated, and Obfuscated Malicious Code*, In Proceedings of the ACM workshop on Rapid Malcode, Oct. 2003.

[13] Linag, Sekar, *Fast and Automated Generation of Attack Signatures: A basis for Building Self-protecting Servers*, In proc. 12th ACM Conference on Computer and Communications Security 2005.

[14] Zhenkai Liang, R. Sekar, Daniel C. DuVarney, *Automatic Synthesis of Filters to Discard Buffer Overflow Attacks: A Step towards Realizing Self-healing Systems*, In Proceedings of USENIX Annual Technical Conference,2005.

[15] H. Etoh, *GCC Extension for Protecting Applications from Stack-Smashing Attacks*, IBM Research, http://www.trl.ibm.com/projects/security/ssp/, Apr. 2003.

[16] CERT Coordination Center, http://www.cert.org