Buffer Overflows Attacks & Defense

Suhas Harbola

Student, University School of Information, Communication and Technology, Guru Gobind Singh Indraprastha University, Delhi, India.

Abstract—*Buffer overflows is one of the most common form of security vulnerability. It may lead to an anonymous Internet user to gain control (partial or total) of a server. Mitigating buffer overflow vulnerabilities we can reduce most of the serious security threats. In this paper, we survey the various types of buffer overflow vulnerabilities and attacks, and survey the various defensive measures that mitigate buffer overflow vulnerabilities.*

Key Words: — **Buffer overflow, Stack smashing attack, bound checking, coding practices**.

1. Introduction

A Buffer is a areas of memory set aside to hold data, often while moving it from one section of a program to another, or between programs [18]. Programmer use data stored in temporary area called buffer before manipulating the data in desired format or extracting relevant information from the buffer. At times the size of data exceeds the size of the buffer this if not controlled causes a memory leakage and distort other memory locations. In a buffer-overflow attack, the extra or overflow data sometimes contains some specific instructions or malicious code added by an attacker which may lead to jump to other instructions in the original code and skip certain code blocks or it may also loop in the malicious code. It overruns the buffer's boundary and overwrites adjacent memory locations.

Buffer overflows occur when n bytes are written into a memory area (buffer) of size less than n bytes. If an attacker gains direct or indirect control of what is written into this memory area, she can carry out buffer overflow attack(s). Buffer overflows have been one of the most widely exploited vulnerabilities, and have led to several high-profile successful attacks.

Attacker would use a buffer-overflow exploit to take advantage of a program that is waiting on a user's input. There are mainly two types of buffer overflows: stack-based and heap-based[18]. Some attacks are difficult to perform and the attacker must have good knowledge of the system, Heap-based is a good example, it floods the memory space reserved for a program. Most common type of attack is Stack-based buffer overflows, it exploit applications and programs by using what is known as a stack: memory space used to store user input.

These kinds of attacks enable anyone to take total control of a host, they represent one of the most serious classes security threats. Buffer overflow vulnerability presents the attacker with exactly what they need: the ability to inject and execute attack code[1]. The injected attack code runs with the privileges of the vulnerable program, and allows the attacker to bootstrap whatever other functionality is needed to control ("own" in the underground vernacular) the host computer.

Buffer overflow vulnerabilities and attacks come in a variety of forms, which we describe and classify in Section 2. Defenses against buffer overflow attacks similarly come in a variety of forms, which we describe in Section 3, including which kinds of attacks and vulnerabilities these defenses are effective against. Section 4 discusses which combinations of defenses complement each other. Section 5 presents our conclusions.

2. Vulnerabilities and Attacks

The main objective of of attacker behind the buffer overflow attacks is to either deny (either through DoS or DDoS attacks) computational resources (processing time and memory) to the legitimate user or to steal valuable information by exploiting software vulnerabilities or to run their malicious code with high privilege. In the first case an attacker confuses the software system by overloading memory buffer with meaningless data which may lead crashing the system. While in the second case the attacker loads memory buffer on the target machine with well formatted data so as to overcome security validation and gain master/root access privileges of the target machine. Once the privilege is gained the attacker jumps the control to the location where malicious code is kept and it executed with the privilege of the running program. This malicious code can either be injected or it is already present in the memory. It is important to understand that in either case buffer overflow is the predominant technique adopted by attackers[2].

The malicious code can be injected in either static, stack and heap and buffer overflows can occur in any one or more of these space. The malicious code can also be kept in a remote location. Although the effects are limited by the area in which the overflow occurs. In static spaces global and static variables are stored which are defined before the program

execution and are not deleted. They have mainly fixed and contiguous memory address.

The stack stores data and variables that are allocated and de-allocated as the process executes. As a function is called its variables and other data will be added to the stack thus increasing the size of the stack. When a function returns all the associated variables and other data leaves the stack and the stack size shrinks. In most systems, return addresses, processor status information, and call frame pointers are also placed on the stack. Return addresses holds the address of the next instruction.

The heap is the space where dynamic allocation is done by the program. Dynamically loadable modules are often loaded into the heap and then executed. Once the memory is allocated in the heap it remains with the variable until the process de-allocates it.

The attacker's goal is to somehow update the variables, return addresses, or function pointers. Variables and function pointers may be modified by overflows in any area while the return addresses can be modified only on the stack. To change the flow of the program and run some malicious code with the privilege of the program, return addresses and function pointers are altered, that may also lead to system crash DoS or DDos. Changing variables means wrong data and it may lead to change in the flow if the variable is used in a conditional expression. This suggests two broad classes of buffer overflow attacks.

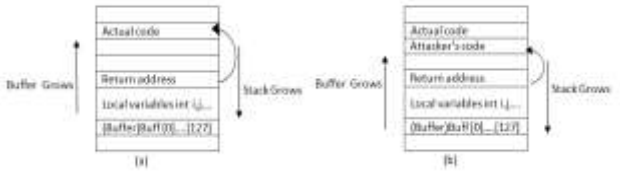


Figure 1: (a)Normal stack processing

(b)Buffer Overflow Attack

2.1 Data Buffer Overflow

A data buffer overflow occurs when a large input overwrites existing data, causing the program to act in a manner that violates security or changes the flow. Let's say that an array and a variable is allocated such that variable is just above the array. If there is an overflow in the array it will change the contents of the variable. The program uses that value and gives access to the unauthorized user. This is direct data buffer overflow. In case program behavior depends on the variable or the variable in turn changes the control variable of the program leading to change in the flow of the program. It is called indirect data overflow. Attacks that change pointers to refer to uploaded data fall into this class.

2.2 Executable Buffer Overflow/ Stack smashing attack

The attacker injects the malicious code into buffer and alters the function pointer or the return address with the address of the injected code so that the control jumps to the malicious code and it gets executed. The injected code is mainly machine understandable code hence the attacker must have prior knowledge of the architecture and op-codes of the machine.

Stack smashing attack [19][2] The attacker modifies the return address or function pointer so that they point to the attacker's code. This results in change in execution flow to the attacker's code when the corrupted information is restored and used by the program counter. In some other cases the attacker uses these critical variables to modify the contents of GOT entries so that the attacker's code is loaded when dynamically loaded library functions are used. These attacks are possible due to the presence of buffer overflow vulnerability in the program.

2.3 Placement new for C++

"Placement new" facilitates placement of an object/array at a specific memory location. When appropriate bound checking is not in place, object overflows may occur. Such overflows can lead to stack as well as heap/data/bss overflows, which can be exploited by attackers in order to carry out the entire range of attacks associated with buffer overflow [3].

```
void *operator new (size_t,void *p) throw() {return p;}
void *operator new[] (size_t,void *p) throw() {return p;}
```

The first construct is used for allocating a single object/data structure, and the second one for arrays. "Placement new" expression makes it possible for the programmer to "place" an object or a dynamically allocated buffer/data structure at a specific memory area. The starting address of this specific memory area is passed as a void * to the new operator. An example of the use of this expression is as follows. Newtext is dynamically allocated 10 bytes (sizeof(char) is one byte) starting at an address that is the value of text. The address must be a non-null one[3].

```
char *text = new char(10);
//Place newtext at the starting address of "text"
char *newtext = new (text) char(10);//uses placement-new
```

It allows any address allocated to the process to be used to place an object. It does not enforce any bounds checking. Neither compile-time or runtime enforcement of bounds checking is applied[3].

```
char c; int *b = new (&c) int;
```

3. Defences

The basic approach to protect from buffer overflow attack is Secure coding. Writing correct and secure code which checks the length of the buffer and the input before trying to fill the buffer, this has to be done by the developer himself. The

other approach can be to make the area that stores stack and other data elements as no executable this can be achieved by making changes to Operating system, even if the attacker injects malicious code the system will not execute that code as it is in no executable region. But the attacker does not necessarily need to inject code to a buffer overflow attack as code can be executed remotely also. Third approach can be to array bounds each time an array is accessed, a compiler can be designed in a way to do this task. This will never let a buffer overflow. But the approach may be a bit expensive. The fourth approach can to detect an overflow and alert the system about it before exiting. Further various methods of detecting an overflow can be classified into algorithmic centric or key centric.

3.1 Coding Securly

One of the basic but expensive thing is to write correct code, a code without any error leading to buffer overflow. Making the developer responsible. The developers (mostly new ones) needs to be trained to counter these kind of attacks and bring in their practice the secure ways of programming. One should consider using safe code like use strncpy instead of strcpy.

Unsafe: char buf[1024]; gets(buf);

Safe: fgets(buf, 1024, stdin);

We can have tools to check for bugs and static analysis of the code can be done to minimize the mistakes. Program fuzzy can be used which will give random values to the variables and errors can be deducted while execution of the program with those values but this becomes more complicated if the function is big an contains many branches.

3.2 Operating system based

Non Executable stack provides for defense against buffer overflow attacks by disabling execution of instructions from stack[2]. This is done by telling the CPU via page protection flags that it's not allowed to execute instructions from stack. One has to apply a special patch to the OS. Drawback is that it resukts in reduced functionality as signal handling in Linux system is done by placing executable code on the stack and gcc uses executable stacks for function trampolines for nested functions. The attacker can place the code in the heap or at a remote location and just overwrite the return address or function pointer to point to the attack code.

3.3 Checking Array Bound

Injection of a code is not necessary for a buffer overload attack thus a non executable storage does not completely stop a buffer overload attack. It fails when the code is already present in the program or is at remote location and the attacker is able to alter the return/function pointers to that code. Not letting the array buffer to overflow will completely stop these kinds of attack. This technique is very time consuming.

[2] To implement array bounds checking, then all reads and writes to arrays need to be checked to ensure that they are within range. The direct approach is to check all array references, but it is often possible to employ optimization techniques to eliminate many of these checks. There are several approaches to implementing array bounds checking, as exemplified by the following projects.

3.3.1 Compilers

The C and C++ compiler can be created or the existing ones can patched to check the array bounds for each array access. As this is a time consuming process optimization can also be employed. Few such examples are discussed.

3.3.1.1 Compaq C

The Compaq C compiler for the Alpha CPU supports a limited form of array bounds checking when the "-check_bounds" option is used. The bounds checks are limited in the following ways:

- only explicit array references are checked, i.e. "a[3]" is checked, while "*(a+3)" is not
- since all C arrays are converted to pointers when passed as arguments, no bounds checking is performed on accesses made by subroutines
- dangerous library functions (i.e. strcpy()) are not normally compiled with bounds checking, and remain dangerous even with bounds checking enabled.

Because it is so common for C programs to use pointer arithmetic to access arrays, and to pass arrays as arguments to functions, these limitations are severe. The bounds checking feature is of limited use for program debugging, and no use at all in assuring that a program's buffer overflow vulnerabilities are not exploitable[1].

3.3.1.2 Bounds checking for C and C++

This project added code to the GNU Compiler Collection to provide run-time checking pointer and array accesses for various bounds errors in compiled code. The primary objectives were to handle C++, to avoid changing the ABI (so that checked and unchecked code can be freely mixed), and to avoid throwing errors on correct code[5].

3.3.2 Hardware bounds checking

The safety added by bounds checking necessarily costs CPU time if the checking is performed in software, however if the checks could be performed by hardware then the safety can be provided "for free" with no runtime cost. Research started since at least 2005 regarding methods to use x86's

built-in virtual memory management unit to ensure safety of array and buffer accesses. In 15 Intel provided their Intel MPX extensions in their Skylake processor architecture which stores bounds in a CPU register and table in memory. As of early 2017 at least GCC supports MPX extensions[4].

3.3.3 Type-Safe Languages.

The buffer overflow vulnerabilities is present in C and C++ because they lack type safety and provide direct memory access. If only type-safe operations can be performed on a given variable, then it is not possible to use creative input applied to variable foo to make arbitrary changes to the variable bar[1]. If new, security-sensitive code is to be written, it is recommended that the code be written in a type-safe language such as Java or ML. Unfortunately, there are millions of lines of code invested in existing operating systems and security-sensitive applications, and the vast majority of that code is written in C. This paper is primarily concerned with methods

3.4 Detecting and Preventing the overflow

3.4.1 Pax Address Space Layout Randomization (ASLR)

The attacker who want to take control of the system using buffer overflow needs to know the address of critical information like return address, saved frame pointer and pointer variables on the stack (among others).It depends on the ability of the attacker to gain this information. ASLR randomizes the base address of the different sections of the program memory (stack, heap, code, data and memory-mapped segments) at load time thus making it difficult for the attacker to know the address of the target object. By randomizing the memory address of critical information on the stack, ASLR breaks down the absolute address assumption made by buffer overflow attack. It is the most widely used technique in several commercial software products, it is susceptible to brute force attacks and in the last few years several enhancements have been proposed [16].

3.4.2 Address space layout permutation (ASLP)

It is similar to ASLR. PaX ASLR randomly relocates the stack, heap, and shared library regions with kernel support, but does not efficiently randomize locations of code and static data segments.In addition to randomization; it permutes the order of functions in the code segment and the order of data in the data segment[17][2]. This is done in two ways. First, we create a novel binary rewriting tool that randomly relocates static code and data segments, randomly re-orders functions within code segment, and data objects within data segment. Our rewriting tool operates directly on compiled program executable, and does not require source code modification.

We only need the relocation information from the compile time linker to perform the randomization rewriting. Such information is produced by all existing C compilers. Second, to randomly permute stack, heap, and memory mapped regions, we modify the Linux kernel. Our kernel changes conserve as much virtual address space as possible to increase randomness. This allows ASLP to support randomization data much finer level than ASLR. ASLP performs randomization at compile time and is implemented by modifying the compiler and linker, where as in ASLR, randomization is done at load time and implemented by modifying the kernel. Unfortunately the published implementation randomizes only the base addresses of code segment and code segments.Hence, it gives no greater security than ASLR against derandomization attacks.

3.4.3 Address space randomization (ASR)

This is similar to ASLP. In addition to permutation of variables, it permutes the order of objects in code and static data segments. It also introduces random gaps between objects (randomly pad stack frames or malloc()'ed regions) [2]. This approach relies on source code transformation tool to perform the randomization. It cannot protect against corruption of nonpointer data as well as pointer-valued data.

3.4.4 StackShield

StackShield[7] copies the return address to a separate memory space thus protecting it from the overflow attack. The saperate memory space is called retarray which is a non-overflowable (a write protected) area.When the function returns the address is restored from retarray. But there are methods to trick this as shown in [8], [9].

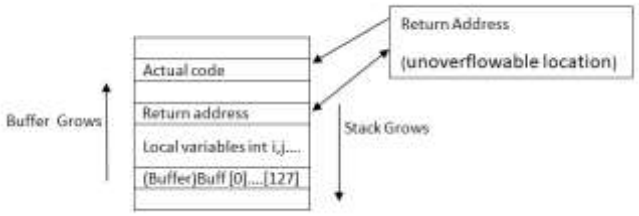


Figure 2: StackShield

3.4.5 StackGuard

StackGuard [1] is a compiler technique for providing code pointer integrity checking to the return address in function activation records. StackGuard is implemented as a small patch to gcc that enhances the code generator for emitting code to set up and tear down functions. It places a "canary"4 before the return address on the stack. While returning the canary is verified before moving on to the return address if any change is found the execution or jump to return address is hault.The canary can be a random value (Random Canary)or a deterministic value(Terminator Canary). Bypassing of this technique is explained in [8],[9].

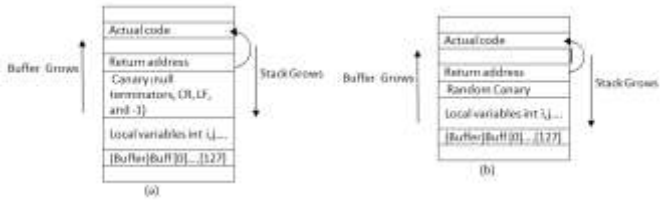


Figure 3: (a) Terminator Canary(b) Random Canary

3.4.6 PointGuard

PointGuard defense against pointer corruption consists of encrypting pointer values in memory and only decrypting the pointers when they are loaded into CPU registers. PointGuard sits between the CPU level 1 cache and registers, it is very important that PointGuard be fast[10].

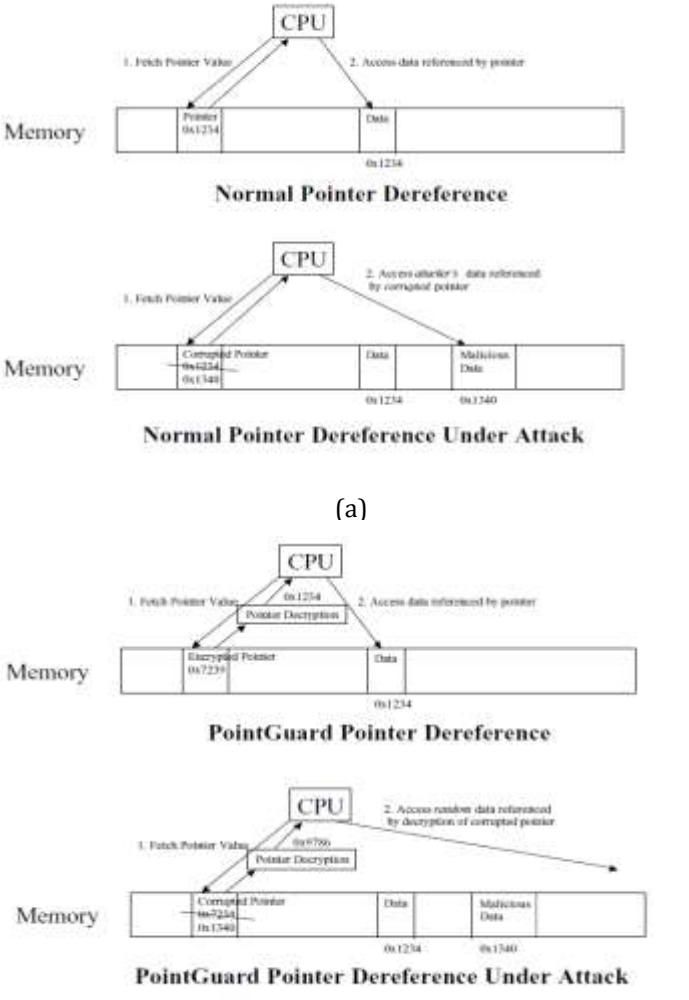


Figure 4: (a) Normal Pointer Attack (b) PointGuard[10]

3.4.7 StackArmor

StackArmor provides an implementation for load time randomization technique. StackArmor performs inter-frame randomization by permuting the layout of the stack frames in the memory. The solution while implemented into the compiler relies on the control flow graph of the program. The major drawbacks are lack of intra-frame randomization and dependency on the need for an accurate control flow graph of the program binary[6].

3.4.8 Instruction Set Randomization (ISR)

Instruction Set Randomization (ISR) is a general approach that defeats all types of remote code-injection regardless of the way it was injected into a process[11]. It create new instruction sets for each process executing within the same system. Code-injection attacks against this system are unlikely to succeed as the attacker cannot guess the transformation that has been applied to the currently executing process[12]. For encoding the machine instructions of the process a key is used, encoded instructions are stored in the processor memory and when the instructions are passed to the CPU decoding is done using the same key. It operates by randomizing the instructions that the underlying system "understands", so that "foreign" code such as the code injected during an attack will fail to execute. If the attackers had access to the machine and the randomized binaries through other means, they could easily mount a dictionary or known-plaintext attack against the transformation and thus "learn the language"

3.4.9 Randomized instruction set emulation (RISE)

Randomized instruction set emulation (RISE) a technique that deliberately obscures the standardized machine instruction set using a private randomized scrambling mechanism [13]. The scrambling function is designed so that it is infeasible to create code sequences to perform a desired function (e.g., an attack) without access to a long secret key that is unique to each program execution.[2] It uses the secret key to scramble the binary using a randomizing loader and the scrambled version is stored in the emulator memory. Then, during the instruction fetch cycle, it unscrambles the fetched instructions, yielding unaltered machine code runnable on the physical machine. It does not protect against attacks that depend on data only attacks. RISE aims to protect the system from binary code injection attacks (injected over the network) into an executing program. RISE and ISR need the support of a virtual environment and incur significant overhead in terms of program execution time.

3.4.10 Data space randomization (DSR)

Data space randomization (DSR) The basic idea behind DSR is to randomize the representation of different data objects. One way to modify data representation is to xor each data object in memory with a unique random mask

("encryption"), and to unmask it before its use ("decryption").It is easy and much simpler to implement and does not need any virtual environment[14].

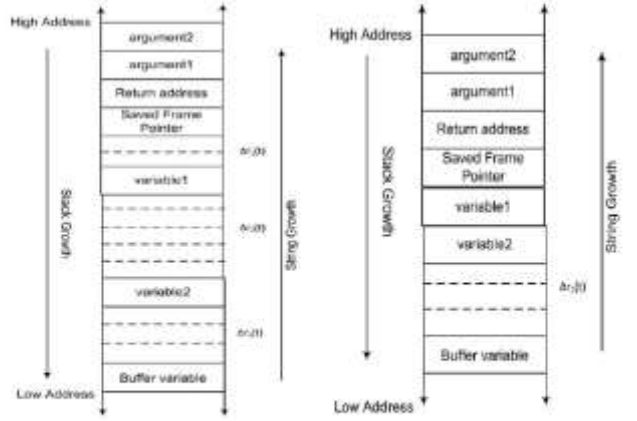
3.4.11 Data Structure Layout Randomization(DSLR)

Data Structure Layout Randomization(DSLR) [2] randomizes the relative distances between program data structures (struct, class, and stack variables declared in functions) and also performs permutation of data structures. DSLR is implemented in the compiler by adding new keywords (obfuscate, reorder, garbage). It provides flexibility by allowing the user to choose the above keywords in any combination. The major limitation of this technique is that the number of possible permutations is a function of the number of the data structures (represented by n) and the number of data elements in each data structure (represented by m). The total number of possible combinations is given by formula (m!)n. In reality this is not a large number as most functions take only 3-4 arguments.

3.4.12 Function Frame Runtime Randomization

FFRR technique transforms the stack by adding random number of words before the memory is allocated to the local variables in the stack. The representation of the random numbers is taken as function of time. The numbers are chosen priori by the function prologue using LFSR the most recent random number is retained and serves as a seed for next random sequence generation.

This approach increases the cost for each function call. To reduce the cost the approach is applied only for buffer-type local variables. For each function we bring in the number of random variables (for each buffer in the function) and the random values are used to add the random number of words (padding) before the each buffer-type local variables. FFRR does not impact the existing process for pushing software updates or patches as the proposed technique randomizes only the run time copy of the program binary[2].



(a) FFRR technique for all local variables (b) FFRR technique for only buffer-type variables

Figure 5: Function Frame Runtime Randomization [2]

3.4.13 Heap Memory Randomization

Heap memory randomization is done by over-allocating the requested chunk of memory in the heap and then placing the returned chunk within the over-allocated chunk. The library functions for heap memory management are wrapped with randomization code. There is no need of source code change of the application as the patches are applied to heap memory management mechanism.

A dual random padding strategy(appending a random pad below and above the pointer to the heap memory chunk) is used for every memory allocation. The randomization ensures that each heap allocation request gets a different buffer during separate instances of running program. The internal layout of the heap chunk is different as the pad1 and pad1 are different on every run. This approach mitigates heap overflow attacks in deployed software [20].

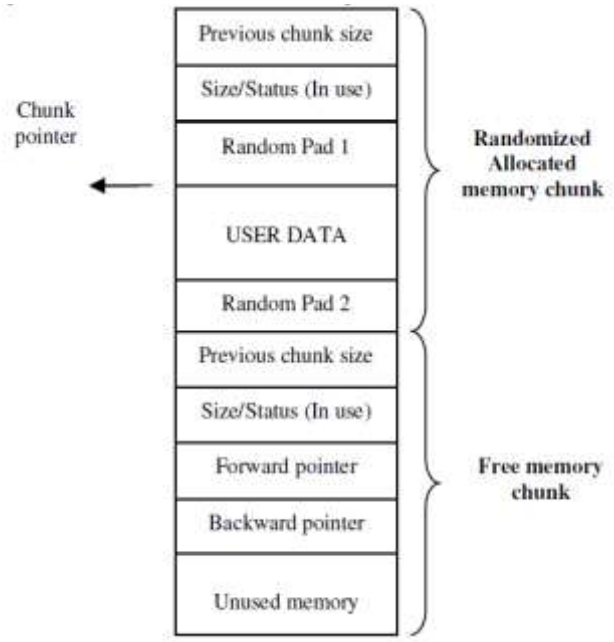


Figure 6: Allocated heap memory chunk with dual random padding [20]

4. Conclusions

We have presented a detailed categorization and analysis of buffer overflow vulnerabilities, attacks, and defenses. Buffer overflows are worthy of this degree of analysis because they constitute a majority of security vulnerability issues, and a substantial majority of remote penetration security vulnerability issues.

5. References

[1] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks," Proc. 7th Conf. on USENIX Security Symp., 1998.

[2] K. S. Kumar and N. R. Kisore, "Protection against Buffer Overflow Attacks through Runtime Memory Layout Randomization," 2014 International Conference on Information Technology, Bhubaneswar, 2014, pp. 184-189.

[3] A. Kundu and E. Bertino, "A New Class of Buffer Overflow Attacks," 2011 31st International Conference on Distributed Computing Systems,Minneapolis,MN,2011,pp.730-739.

[4] https://en.wikipedia.org/wiki/Bounds_checking

[5] Bounds checking for C and C++ http://www.imperial.ac.uk/pls/portallive/docs/1/18619746.PDF

[6] Design of Experimental Test Bed to Evaluate Effectiveness of Software Protection Mechanisms Against Buffer Overflow Attacks Through Emulation.pdf

[7] StackShield. http://www.angelfire.com/sk/stackshield/

[8] G. Richarte, "Four different tricks to bypass stackshield and stackguard protection," Core Security Technologies.

[9] Kil3r Bulba, "Bypassing StackGuard and StackShield," Phrack, vol. 10, no. 56, 2000.

[10] C. Cowan, S. Beattie, J. Johansen and P. Wagle, "PointGuard: Protecting pointers from buffer overflow vulnerabilities," Proc. 12th USENIX Security Symp., pp. 7-17, 2003.

[11] Fast and Practical Instruction-Set Randomization for Commodity Systems Georgios Portokalidis and Angelos D. Keromytis Network Security Lab Department of Computer Science Columbia University, New York, NY, USA {porto, angelos}@cs.columbia.edu.

[12] S.Kc. Gaurav, A.D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," Proc. 10th ACM Conf. on Computer and Comm. Security, pp. 272-280, 2003.

[13] E.G. Barrantes, D.H. Ackley, T.S. Palmer, D. Stefanovic, and D.D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," Proc. 10th ACM Conf. on Computer and Comm. Security, pp. 281-290, 2003.

[14] S. Bhatkar and R. Sekar, "Data space randomization," Proc. of the 5th Int'l Conference on Detection of Intrusions and Malware & Vulnerability Assessment, pp. 1-22, vol. 5137 of Springer Lecture Notes in Computer Science, 2008.

[15] Z. Lin, R.D. Riley, and D. Xu, "Polymorphing Software by Randomizing Data Structure Layout," Proc. 6th Intl Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 107-126, 2009.

[16] Pax, "ASLR," http://www.pax.grsecurity.net/, 2003.

[17] C. Kil, J. Jun, C. Bookholt, J. Xu and P. Ning, "Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software," Proc. Ann. Computer Security Applications Conf., pp. 339-348, 2006.

[18] https://en.wikipedia.org/wiki/Buffer_overflow

[19] Aleph One, "Smashing the stack for fun and profit," Phrack, vol. 7, no. 49, 1996.

[20] V. Iyer, A. Kanitkar, P. Dasgupta and R. Srinivasan, "Preventing Overflow Attacks by Memory Randomization," 2010 IEEE 21st International Symposium on Software Reliability Engineering, San Jose, CA,2010,pp.339-347