**Computer Architecture for Security:**

**Buffer Overflow Attacks and How to Secure Computer Systems Against Buffer Overflow Attacks**

By

Cecile Sano

Instructed by Professor Morris Lancaster

Course Term Paper

Submitted in partial fulfillment of the requirements

for the CSCI 6461 Computer System architecture course

in the Graduate Program of the

George Washington University, 2015

Washington, D.C.

INTRODUCTION

**Hardware and Software**

The goal of compromising software is to abuse computing resources for various purposes, such as gaining illegitimate access to data and stealing sensitive information. The buffer overflow is widely known as the most commonly exploited vulnerability targeting for memory-based attacks in software applications [1].

Hardware can improve the speed and reduce the overhead of buffer overflow prevention by implementing security mechanisms with smaller performance penalties than those implemented by software [1]. A secure hardware can also provide a trusted base for computing and can mitigate the software vulnerabilities by protecting an application from software-based attacks [1]. A secure system can be built only from a secure and a trustworthy hardware.

**Buffer Overflow**

Security vulnerabilities are discovered in many software every week [2]. It is a well known fact that currently, buffer overflow is the most common exploit in the world of computer security and that it is one of the costliest security vulnerabilities affecting computer software [3]. Using buffer overflow exploitation, a malicious user attempts to gain control of a computer sys­tem by overwhelming it with skillfully designed input data [2]. Buffer overflows have been detected in many types of software ranging from Web browsers to Web servers [2]. Software such as Internet Explorer, Hypertext Preprocessor (PHP), and Apache have all been victim to such vulner­abilities [2]. Because of the widespread availability and use of vulnerable software, buffer overflow exploits can be a seri­ous threat to system and data integrity, and in many ways could cause significant problems in software systems [4]. For example, it could cause severe threats to humans or to the economy by compromising critical systems such as health-care, nuclear or aerospace software applications [4]. Attackers can use buffer overflows to launch denial-of-service (DoS) attacks, spawn a root shell, gain higher-order access rights such as administra­tor privileges, steal sensitive information, eavesdrop, or impersonate a user [5]. Although, most of these vulnerabilities are detectable at compile time, few compilers provide such capabilities [2].

A buffer overflow occurs when a program or process tries to store more data in a buffer, which is an area to temporarily store data, than it is intended to hold [6]. Since buffers could contain only a finite amount of data, the extra information, which has to go somewhere can overflow into adjacent buffers, corrupting or overwriting the valid data held in them [6]. These extra data could contain attacker's malicious codes that are designed to trigger specific actions such as damaging the user's files, changing data, or disclosing confidential information [6]. The first buffer overflow attack was found in 1988 as an exploitation of a flaw in systems based on BSD versions of UNIX [7]. This allowed an internet worm program known as the MORRIS worm to break into those BSD machines and to spread from one machine to another [7]. Eventually the worm spread to thousands of machines and interrupted the Internet connectivity and other normal activities for many days [7].

In July 2000, a programming flaw made it possible for an attacker to compromise the integrity of the target computer by simply sending an e-mail message through Microsoft Outlook and Outlook Express [6]. The programs' message header mechanisms had a defect that made it possible for senders to overflow the buffer with extraneous data, which allowed them to execute malicious code on the recipient's computers [6].

Other attacks include the Code Red Worm which exploited a buffer overflow vulnerability in Microsoft IIS webserver in 2001, infecting 359,000 systems within 14 hours [8]. In 2003, SQL Slammer exploited a buffer overflow in the Microsoft SQL server and launched a DoS attack on various targeted networks [9]. It spread quickly and caused serious network outages and consequences such as canceling airline flights, interfering with elections, and failing ATM machines [9]. Another buffer overflow attack known as modchips allowed unlicensed software to run on the Xbox console without any hardware modifications by taking advantages of a flow in licensed games developed for the consoles [7]. The National Vulnerability Database recorded 176 buffer overflow vulnerabilities, of which 136 had a high severity rating in the first five months of 2010 [10].

David Kramer states in his article *Buffer Overflow* that "Buffer overflow attacks are said to have arisen because the C programming language supplied the framework, and poor programming practices supplied the vulnerability [6]." Buffer overflows resulted from the implication of poor programming practices, meaning, not applying any boundaries on the input size the program can handle [3]. C and C++ programming languages do not include built in boundary checking, which allows direct access to application memory [7]. In general, buffer overflows cause the system to crash. However, with a well planned overrun, an attacker can get execution control without making the system to crash [7]. Although many overflows occur when the program receives more data than it expects, there are many different kinds of overflow attacks.

According to US-CERT Vulnerability Notes Database, on an average, every third computer security attack is based on buffer overflow exploits [7]. In addition, buffer overflow was ranked third on the Common Weakness Enumeration and SANS listed it as one of the top 25 most dangerous software errors in 2011 [11]. Therefore, it is important to well understand the principle of buffer overflow exploits and to provide effective protection and defensive techniques against these attacks.

PRINCIPLE OF BUFFER OVERFLOW

**Background on Computer Architecture**

An executing computer program is made up of three main memory areas, the instruction memory, the stack, and the heap [2]. The instruction memory contains machine code that is executed by the Central Processing Unit (CPU) [2]. The stack area is composed of Activation Records (AR), also known as stack frame, which is created for each func­tion call and stores information such as return address and local variables (See Figure 1) [2]. The stack uses temporary memory. The heap is the area of memory used by programs to dynamically allocate memory [7]. The heap does not use temporary memory and it is expected to remain in use while a program runs [12].

The Instruction Pointer (IP) is a processor register which contains the address of the next instruction to be executed [2]. When a func­tion is called, an AR containing copies of this address is created, pushed onto a stack, and the caller function places the current IP on the stack in reverse order along with the function arguments [2]. When the callee receives the execution command, it saves the caller's stack pointer and then allocates memory for local variables [2]. By using pointers, a programmer can locate the local variables in stack memory and hence can overwrite the IP used to return execution back to the function caller [2]. When a function completes execution, this IP is set back to the previous address on the stack where execution resumes [2]. Thus, if the stored IP is overwritten, execution can resume at any place in memory depending on that modification [2]. In fact, the saved IP can be easily over­written with simple pointer arithmetic [2]. By adjusting the saved IP to point back to the code invoking the function call instead of resuming after it and then exiting, an attacker could modify the program in such a way as to make it endlessly reprint the string [2].

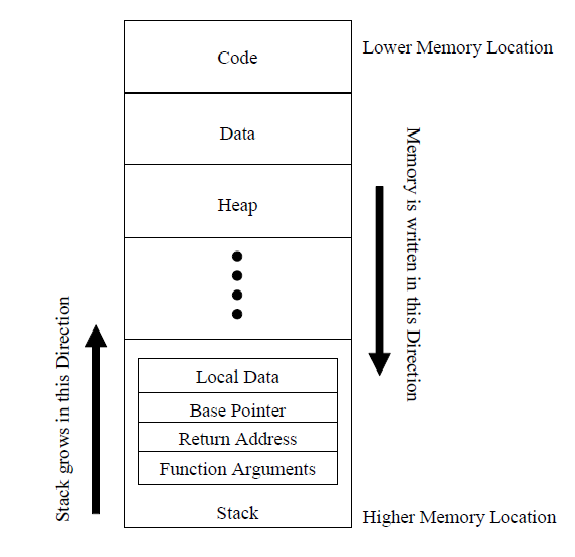


Figure 1. Activation Record (AR) [7]

**Buffer Overflow Exploits**

Buffer overflow exploits generally target function activation records, pointers, return addresses or the management data. To carry out an exploit, attackers must find suitable code to attack and make program control jump to that location with the input data in memory and registers [5]. When the program is compromised, it will not always crash and it will also not always behave as advertised [13]. A hacker noticing this behavior then tries to insert various inputs until he finds a way to corrupt the stack or the heap and then executes malicious code [13]. Here is how a buffer overflow exploit works [14]:

1. Fill the buffer with correct machine code.
2. Overlay the return address with the buffer’s starting address.
3. Use the return address to control the program jump to the system stack in order to execute the desired/malicious code.
4. Gain direct control of the system and do any desired operation to it.

Figure 2 below shows the difference between before and after the buffer overflow attack.

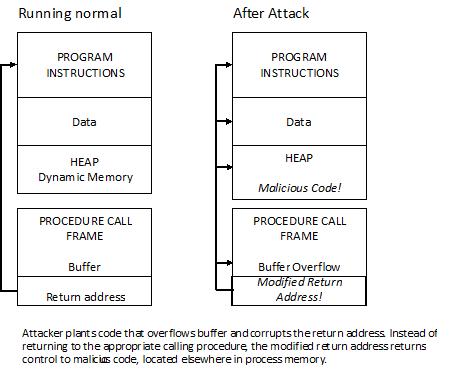


Figure 2. Buffer overflow attack [15]

Sometimes hackers find other ways to exploit the overflow besides getting their code to run. Certain overflows do not actually allow hackers to take control, but might instead allow them to manipulate data [3]. One way to categorize buffer overflow attacks is [7]:

1. Stack Buffer Overflow

2. Heap Overflow Attacks/Corruption

3. Format String Attacks

4. Others attacks

The ultimate goal of every buffer overflow attacks is to get execution control [7]. There are different targets and ways to overwrite and different ways to start executing malicious code to achieve this ultimate goal [7]. The techniques to exploit buffer overflow vulnerabilities can also vary depending on architecture, operating system and memory region [7].

**1. Stack Buffer Overflow**

Stack buffer overflows occur when more data is written to a buffer stored in a stack, which stores temporary information, than the allocated memory space [3]. In a stack-based buffer overflow attack, the attacker adds more data than expected to the stack [12]. If the local data stored inside a character buffer within a stack exceeds its allocated space, then it can overwrite the return address or base pointer or other data at higher memory locations, or any combination of all of them [7]. Hence when function call returns, instead of returning to the caller function, it jumps to other location written by the attacker, and resumes execution from that point [7]. This is when the attacker's commands fall into the stack, allowing the program to follow instructions that it would not normally do. If the instructions are not perfect, the program crashes but if the instructions are perfect, the program carries out the attacker's instructions [12].

*Off By One Attack*

In programming language like C and C++, character buffers are supposed to have a terminating null character [7]. However since programmers often forget about it, even though there is a boundary checking, many buffer operations will not add this character, leaving no border between two consecutive buffers [7]. As a result, both consecutive buffers could get treated as a single entity [7].

If there is some boundary checking on a buffer, it is not possible to overflow the buffer far enough to overwrite the return address (Register EIP) [7]. However, it is possible to partially overflow the buffer and gain execution control by just modifying the base pointer (Register EBP) [7]. As shown in Figure 3, if an attacker overwrites EBP then the interpretation of caller function’s AR can be altered, which means that when the called function returns, it will return to the new AR pointed by the new EBP [7]. Since all execution instruction use EBP’s value as a reference to compute stack top and return address, by compromising EBP, an attacker is also changing values of ESP (the current stack pointer) and EIP, and hence can compromise the system control [7]. This kind of attack is called "off by one" attack.

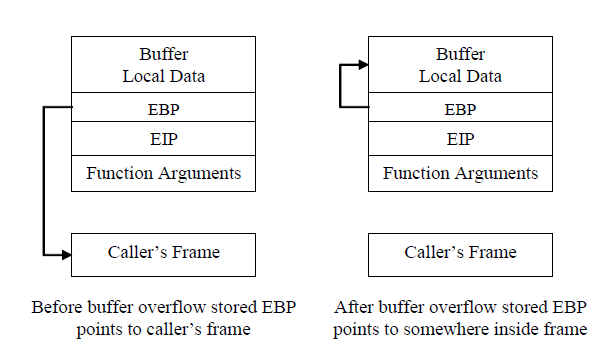


Figure 3. Off by one attack [7]

**2. Heap Overflow Attack (Heap Corruption)**

Conceptually, the heap overflow attack is the same as the stack attack, except that it occurs in the heap instead of the stack. The goal is to overwrite function pointers on heap to point to injected code [16]. Heap overflows can be very tricky as heap rolls back to the previous AR when the called function is complete [7]. Pages in the heap can be read from and written to, hence attackers take advantage of this by writing attack instructions into the pages, tricking the computer into following these instructions [12]. The heap contains vital information that can be exploited to get execution control such as function pointers and dynamic memory allocation metadata used by allocation functions [7].

**3. Format String Attacks**

Format string attacks usually add a single address in memory that points to another address in memory where the attacker has added new instructions to execute [12]. It happens when an application uses user supplied data on a function as part of a format string argument, such as printf() function [7]. The problem arises when programmers fail to prevent non-trusted externally supplied data from being included in the format string argument [7].

**4. Others attacks**

*Integer overflow* occurs when a specific data type of CPU register meant to hold numeric values within a certain range is assigned a value too large to be represented within that range [3]. Integer overflow often leads to a buffer overflow [3].

*Stack-smashing attack* is the most basic and easiest form of a buffer overflow attack [17]. This attack writes data on the stack to disrupt normal program execution [2]. In the best case, any extra input may cause a memory access fault, but in the worst case, it overwrites program data [17]. There is little precision involved in how the stack’s memory is overwritten, all nearby memory locations are simply overwritten with the malicious address and hence it is called stack "smashing" attack [17].

*Pointer subterfuge* involves modifying pointer values, such as function, data, or virtual pointers, and can also modify exception handlers [14]. Attackers can use pointer subterfuge to overflow stacks, heaps, or objects containing embedded function pointers [14].

*Function pointer attacks* are generally much more difficult to accomplish than buffer-overflow attacks since a function must not only define both a function pointer and buffer, but they must also appear with the function pointer following the buffer on the stack [17].

*Longjump buffer attack* is one of the methods for hijacking program control [14]. The C standard library provides setjmp/longjmp to perform nonlocal jumps. Function setjmp saves the calling function’s environment into the jmp\_buf type variable, which is an array type for later use by longjmp [14]. Longjmp restores the environment from the most recent invocation of the setjmp call and an attacker can overflow the jmp\_buf with the address of the attacker’s code [14]. When the program calls longjmp, it will jump to the attacker’s malicious code [14].

*Return-to-libc attack* does not need an executable stack nor a shell code that a common buffer overflow exploit use [18]. Instead, it causes the program to jump to some existing code, such as the system() function in the libc library which is already loaded into the memory [18]. There are many more attacks, such as Heap smashing, and Arc injection but I will not go over them in this paper. In all cases, these more sophisticated attacks maintain the idea of stack smashing, targeting addresses and disrupting the expected flow of execution in a program [17].

BUFFER OVERFLOW DEFENSIVE TECHNIQUES

The best defense against any of these buffer overflow attacks is writing a perfect and secure code. However, programmers are often more focused on performance over the correctness and security of the code [12]. Therefore, programs are not perfect and bugs in the program permit buffer overflow attacks [12].Many methods have been used to prevent dam­age caused by buffer overflow attacks. These methods usually fall into one of two categories, proactive and reactive [2].

Proactive defenses prevent buffer overflow. This type of defense usually involves checking every memory read/write and ensuring it is done within the proper memory area [2]. Although this technique is highly effective against stack smashing, it causes program slowdown [2]. One good example of proactive defenses is Boundary Checking. The concept of a boundary checking is that ideally, every field in every program would allow only a given number of expected characters [12].

Reactive defenses allow a program to write anywhere in memory as it normally would, but prevent undesirable program execution [2]. These defenses usually involve validating memory at the end of a function call to check if the saved IP (Instruction Pointer) or other parts of the stack have been overwritten [2]. If buffer overflow is detected, the program exits or begins executing a recovery routine [2]. Examples of reactive defenses include the following:

StackGaurd is a software tool which introduces a “canary” byte next to the return address on the stack, and was developed by a group of Linux developers [12]. Thus, an attacker must overwrite this byte along with the saved IP in order to cause a buffer overflow [2]. At the end of a function call, StackGaurd checks to see if this byte has been overwritten [2].

StackShield is a software tool that exploits pointer copying approach [19]. It stores and maintains copies of return addresses in the memory area of the program and checks the integrity on each function call [19]. When a function returns, the program checks to see if the IP in the function’s AR differs from the copied version [2]. However, copied return addresses can be manipulated by data-pointer modification attacks [19].

Hardware-assisted approaches to buffer overflow protection improve accuracy and performance of schemes involving software for dynamic attack detection [1]. One common solution is to maintain a shadow of the return address in hardware by creating a return address stack or monitoring the location of the return address for any unauthorized modifications [1].

Current solutions attempt to prevent at least one of the three steps of buffer overflow attacks [19]: (1) Buffer overflows themselves (e.g. boundary checking approach). (2) The modification of code pointers (e.g. integrity check of code pointers). (3) The execution of injected code (e.g. execution prevention approach).

**Secure Coding and Boundary Checking**

The main cause of buffer overflow attacks is the insufficient boundary checking on arrays [7]. The main methods of array bounds checking include compiler checking, memory access checking and using safe programming languages [14]. As mentioned before, languages like C and C++ do not have built-in protection for detecting out-of-bound memory accesses [14]. However, newer and safer versions of C/C++ compilers that come with safe libraries such as strcpy\_n provides more boundary checking and thus can avoid many potential buffer overflows [7]. Also, using languages like Java or environments like .NET that perform runtime bounds checking will eliminate the problem [14]. Hence, a proper choice of language during program development is very important.

**Runtime Techniques**

Many runtime techniques for defending against buffer overflow attacks use return address modification to detect buffer overflows [14]. Compile-time techniques like StackGuard and Return Address Defender (RAD) insert code to check for return address modification [14]. As mentioned earlier, StackGuard places a canary word before the return address and checks its value when the function returns. RAD creates a Return Address Repository global array and copies the return address to it in the function prologue, then checks for modifications in the function epilogue [14]. However, since attackers can alter the return address indirectly by using a pointer, these approaches could still be vulnerable [14].

Another method to thwart the buffer overflow attacks using runtime techniques is to make critical configuration changes in the Windows environment [3]. Microsoft Visual Studio C++ compiler offers several options to enable certain checks at runtime such as /GS, RTC, Runtime library check and DEP (Data Execution Prevention) [3].

/GS switch causes the compiler to add extra code for detecting various buffer overflows by inserting a security cookie XOR'd with the return address at function entry [7]. This method does not require any modification in source code but requires a re-compilation of the source code [7].

The RTC compiler option controls run-time checks such as underflow and overflow checking, stack verification and detection of variable use without initialization [3].

**Execution Prevention Approach (Non-Executable Stack Methods)**

Execution Prevention is an approach which prevents the execution of code on the stack or on the heap by marking the memory regions as non-executable [7]. If an attacker tries to execute machine code in these regions, this mechanism will cause an exception, which makes the system to halt and thus avoids the attack [7]. It makes use of hardware features such as the NX ("No eXecute") bit which is used to segregate memory areas [7]. It assumes that no code is intended to be executed that is not part of the program itself [3]. Since each page loaded into memory is marked if execution can happen in that page or not, data pages can only be read or written and cannot be executed [7]. UNIX versions like Open BSD come with executable space protection packages like Pax and Openwall that use a non-executable stack to prevent buffer overflow [7]. However, non-executable stack methods cannot defend against return-to-libc attacks and attacks on data segments because some instances also need an executable stack [14]. In addition, this method requires administrative right to change its settings [3].

Software fault isolation (SFI), also called “sandboxing” stops memory modification or prevents execution of instruction by rewriting the un-trusted code at the instruction level in executables [7]. The original SFI technique was applicable only to RISC (Reduced Instruction Set Computer) architectures but recent development represents the same technique for CISC (Complex Instruction Set Computers) architecture as well [7].

**Integrity Checking**

The processor inherently trusts the addresses given to it during the execution of a program and this level of trust is exactly what attackers exploit [17]. By manipulating a code pointer, an attacker can change the control flow of the target program and force it to jump to the injected malicious code [19]. To solve this problem, it must be assured that addresses are trustworthy at all times [17]. Integrity check of code pointers detects and prevents modification of code pointers, especially return addresses so that the control flow of the program cannot be altered by buffer overflow attacks [19]. There are three major categories that fall into integrity check approach [19]: Pointer copying, Pointer encrypting, and Canary Word method.

**1. Pointer Copying**

Pointer copying is a method that implements the integrity checking of code pointers by storing their copies in some memory area in the program's address space [19]. When a code pointer is referenced, the original code pointer is compared with its copy to determine whether the former has been manipulated [19]. Any difference in the two values is detected by the runtime system and further execution is prevented [19]. Pointer copying is a simple and deterministic method, and it is compatible with the existing code because unlike the canary word and pointer encryption, it neither changes the stack layout nor the pointer value [19]. Example of methods that exploit pointer copying approach are: MineZone RAD which places guard pages around the memory area where return addresses are stored, and Read-Only RAD which sets the memory area as read-only by mprotect() system calls [19].

**2. Pointer Encrypting**

StackGhost encrypts the return addresses on the stack to detect a corrupt return address [19]. Even if an attacker manipulates the return address on the stack, it is decrypted to an unexpected address, which unless the key is known will cause a fault and detects the attack [19]. StackOFFence and HSDefender detect the manipulation of the return address by pushing an encrypted return address along with the original one on the stack [19]. PointGuard, a generalization of StackGuard, encrypts all code pointers, including return addresses with a process-specific key [19]. Unfortunately, these pointer encrypting methods are known to be vulnerable to read attacks and replay attacks (intercepting the data and retransmitting them) [19].

Address space layout randomization (ASLR) encrypts addresses instead of pointers by obfuscating memory addresses [16]. In other words ASLR is a method of randomly arranging the positions of important data areas in a memory, mainly the base address of the executable and libraries, stack, and heap [7]. Due to this randomization, it is difficult to find the function addresses and data variables, and hence the exploitation of a buffer overflow is not impossible but becomes much more difficult [7]. This randomization is also local to system and hence the attacker cannot generalize the attack and it foils the attempts of internet worms [7]. However, this approach is still vulnerable to de-randomization attacks [20].

Intrusion Set Randomization (ISR) is a method to obfuscate the instruction set by encoding the machine code of an executable and decoding instructions before sending it to processor [16]. It performs decoding when instructions are fetched from memory [16]. ISR effectively eliminates executing injected code [16].

**3. Canary Word**

Canary Word is one of the most popular defense schemes that protects addresses from the kind of buffer overflow attacks that overwrite the stack [12]. The terminology comes from a Canary bird used for detecting toxic gas in a mine [21]. Since the bird is more sensitive to the toxic gases found in the mine than humans, if the bird dies, this would be a good indication of a fatal condition in the mine [21]. The key idea behind this technique is to detect the buffer overflow in a stack as they occur, assuming that corrupting an address will also corrupt the adjacent data [7]. Therefore, a Canary Word is placed as a metadata, adjacent to the address [21]. It protects the stack by being inserted near return addresses which are sensitive locations in the stack telling the computer where to find the next commands to execute [12]. It detects the manipulation of return addresses by checking whether the canary value is altered [19]. If an overrun takes place, this canary data get altered, which is the evidence of the overflow and the program quits [7].

The weak point of Canary Word is that there is no mechanism to protect the integrity of the Canary Word itself and thus it can be bypassed by data-pointer modification attacks [19]. In addition, Canaries cannot protect against heap attacks since heap attacks do not affect the stack at all, completely avoiding the canary [12]. Although this scheme could also be easily bypassed by replacing a Canary Word with a valid Canary value, this overwrite could be detected with high probability due to the variations of the Canary Words [13]. The most well known schemes using Canary Word are StackGuard, StackGhost and ProPolice [7].

1) StackGuard

StackGuard was the first canary stack protector, released for GCC (GNU Compiler Collection) in 1997 [7]. It doesn’t require any source code change but requires re-compilation [7]. StackGuard writes a Canary Word between the local variables and the return address on the stack before a function is called, and checks it just before the function returns [13].

2) ProPolice (SSP)

ProPolice, also known as "Stack-Smashing Protector", is a successor a of the StackGuard which rearranges the stack layout so that old base pointers and local variables are also protected [19]. In ProPolice canary code generation is done at the front-end of the compiler while in StackGuard it is done at the back-end [7]. ProPolice also protects not only the return address but all registers saved in function's prologue, which provides more security compared to StackGurad [7]. In addition to canary protection, ProPolice also sorts array variables where possible to the highest part of the stack frame, making it more difficult to overwrite any subsequent variables [7]. It also effectively protects the arguments by creating copies of function's arguments and relocating them together with local variables [7]. Nevertheless possible regeneration of the canary word and the need for recompilation are still disadvantages of ProPolice [7].

3) StackGhost

StackGhost, is a patch for the Open BSD kernel for the SPARC architecture [7]. It protects applications' stack frames and mainly guards the return pointers by detecting modifications of return pointers transparently [7]. The protection mechanism does not require source code or recompilation, but instead requires hardware modifications, which restricts the usefulness of StackGhost [7].

**4. Other Integrity Checking Methods**

Secure Bit provides a hardware bit to protect the integrity of addresses for the purpose of preventing buffer overflow attacks [22]. Secure Bit is transparent to the user and can detect and prevent all address-corrupting buffer overflow attacks with little runtime performance penalty [22]. It is associated with each memory word and tracks data (the address) passing across domains such as processor and kernel [21]. The associated instructions such as call, return and jump verify the data before using it [21]. If an attacker overflows a buffer and attempts to overwrite the return address, the memory word holding the return address will have the Secure Bit cleared by the write operation [17]. Then, when the processor attempts to return, it will see the Secure Bit has been cleared and halt execution [17]. This is how Secure Bit protects return addresses.

In addition, several papers have proposed hardware support for checking the integrity of return addresses by implementing a hardware-based return address stack (RAS) feature [19]. A hardware-based RAS usually has a very low overhead and achieves strong protection [19]. However, the downside is that it requires modification to the processor architecture and is difficult to apply to the existing systems equipped with IA-32 processors [19].

**Other Existing Methods**

Deep packet inspection (DPI) is a form of packet filtering over computer network [7]. The key idea is to examine the data and the header in the packet as it crosses examining software [7]. DPI can block packets that have the character of a known attack, or some malicious pattern in data [7]. One good example in blocking a possible attack is by detecting a long series of No-Operation (nop) instructions and flagging them as a possible payload [7].

Secure Return Address Stack (SRAS) implements a shadow stack in hardware with processor modifications including ISA (Instruction Set Architecture) changes, additional logic, and protected storage [1]. Unlike the usual call stack, the shadow stack only holds return addresses [1]. On a function call, the return address is pushed to the regular stack and the shadow stack [1]. On a return, SRAS pops and compares the return address from both stacks [1]. Shadow stacks keep extra copy of return address in separate memory space and only allows a return if the two addresses matches [16]. To handle function call nesting, the OS is modified, and the spill-over of the secure stack is stored in a special part of memory that is accessible only to the kernel [1]. The kernel is responsible for managing a secure spillover stack for each process [1]. The limitations to applying shadow stacks are that they do not protect other data such as local variables and heap overflow, which can overwrite function pointers [16].

Binary of Split Stack (BoSS) is a tool developed in Microsoft Visual C++ 2005 targeted for Microsoft Windows operating systems on Intel machines with a 32bit architecture [7]. It takes .exe or .dll files as input and modifies the file with the new layout which can withstand buffer overflow attacks [7]. BoSS does not depend on the source code and hence applications to be modified by BoSS could be downloaded from the Internet [7].

SmashGuard, prevents spill-over stack growth by modifying the processor and the OS, and by including security functions during context switching [1]. A function call and a return instructions store a return address inside a memory-mapped hardware stack that is checked on the function’s return [1].

MemGuard is a generalization of StackGuard which sets stack pages as read-only in order to protect the return addresses and to install a trap handler to catch and copy to stack pages [19]. It also uses debug registers to cache the protected return addresses in order to reduce traps to the kernel [19]. However, MemGuard still has a high performance penalty due to the frequent entry of a kernel [19].

Secure Cache (SCache) uses cache memory to protect the return address by providing

replica cache lines that shadow the return address [1]. A corrupted return address will have a different value in cache than its replica and hence would be detected [1]. A drawback to using cache space for storing the return address is that performance is sensitive to cache parameters and behavior [1].

CONCLUSION

Buffer overflow attack is the most common and dangerous attack method at present [14]. The goal of a buffer overflow exploit is to disrupt a desired program flow [2]. Specifically, it often attempt to gain entire or partial control of a system. System control is overtaken by overflowing a data buffer and overwriting a nearby saved IP [2].

There are many kinds of buffer overflow attacks and many defensive techniques. Unfortunately, it was impossible to provide every exploits and protection schemes in this short paper. However, this paper provides an understanding of basic buffer overflow attacks, the different types of attacks, and some of the defensive methods and in some cases explains why even these methods do not always work.

As introduced in this paper, there are many ways to circumvent buffer overflow attacks. Many programming methodologies and software tools exist to detect and prevent these vulnerabilities [2]. Buffer overflow prevention schemes could generally be fit in one of the following categories [7]:

* Source code modification in order to make it safer.
* Compiler modification and recompiling the source code with it.
* System or network hardware modification in order to detect a potential attack.
* Modify the executables files of the software.

Many scientists are developing new defensive schemes such as Secure Canary Word and SegmentShield based on the already existing techniques.

Secure CanaryWord combines Secure Bit and Canary Word together [21]. While Canary Word can protect an adjacent address, Secure Bit can provide the integrity for the Canary Word [21]. For this reason, the Secure Bit is used to erase its weakness by detecting an overflow of the Canary Word [21]. Figure 3 shows the difference between Secure Bit and Secure Canary Word [21].



Fig 3 [21]. Secure Bit Secure Canary Word

SegmentShield uses pointer copying as the basic idea and is a strong and efficient protection scheme that exploits the segmentation mechanism of IA-32 (Intel x86) processors [19]. It stores the copied code pointers into a dedicated hardware segment instead of a memory area allocated in the address space of the program [19]. Since an IA-32 segment is an isolated address space that cannot be accessed by a normal pointer in the address space, the copied code pointers are protected from being overwritten by buffer overflow attacks including data-pointer modification [19]. This scheme achieves a low runtime overhead while ensuring strong protection with hardware support [19].

Ultimately, the best defense is not to write code that is exploitable. Although some programming languages enforce more secure code, they still allow unsafe constructs [13]. Implementing C and C++ sources and some of the C library methods that make the program vulnerable into the source code should be avoided.

Nevertheless, the biggest defense against buffer overflow exploits is to prevent them from occurring. Defensive programming techniques such as validating user input, using length-aware functions, pointer boundary checking, and check­ing for null pointers are often the best solutions to protect the system from buffer overflow attacks [2]. Hence, it would also be helpful to run applications at the least possible privilege level and to consider using an input validator [13]. Also, data should be checked whenever it crosses the boundary and only safe string function calls should be used [13].

It is essential to understand the weaknesses in the computer systems and mastering methods of exploits in order to protect them from various buffer overflow attacks. Establishing integrated exploration of program analysis, pattern recognition, and data mining is highly recommended [14].

REFERENCES

[1]: G. Bloom, E. Leontie, B. Narahari, R. Simha. (2012). Hardware and Security: Vulnerabilities and Solutions. Retrieved from http://www.seas.gwu.edu/~simha/research/HWSecBookChapter12.pdf

[2]: Isaac Greg. (2005). an overview and example of the Buffer-overflow exploit (p16-21). Retrieved from http://iac.dtic.mil/csiac/download/Vol7\_No4.pdf

[3]: Ajay Yadav. (April 23, 2014). Buffer-overflow Attack Defense.

Retrieved from http://resources.infosecinstitute.com/buffer-overflow-attack-defense/

[4]: C. Del Grosso, G. Antoniol, E. Merlo, and P. Galinier. (October 2008). Detecting buffer overflow via automatic test input data generation. Retrieved from http://www.sciencedirect.com/science/article/pii/S0305054807000305

[5]: Bindu Madhavi Padmanabhuni and Hee Beng Kuan Tan. (2011). Defending against Buffer- Overflow Vulnerabilities. Retrieved from http://www.ccrc.wustl.edu/~roger/361S.f11/mco2011110053.pdf

[6]: David Kramer. (June 2007). Buffer overflow. Retrieved from http://searchsecurity.techtarget.com/definition/buffer-overflow

[7]: Parag Nileshbhai Doshi. (2007). Preventing Buffer Overflow Attacks Using Binary of Split Stack (BoSS). Retrieved from http://books.google.com/books/about/Preventing\_Buffer\_Overflow\_Attacks\_Using.html?id=I6J0qdyrb\_kC

[8]: David Moore and Colleen Shannon. (July 2013). The Spread of the Code-Red Worm (CRv2). Retrieved from http://www.caida.org/research/security/code-red/coderedv2\_analysis.xml

[9]: David Moore et.al. (2003). Inside the Slammer Worm. Retrieved from http://cseweb.ucsd.edu/~savage/papers/IEEESP03.pdf

[10]: The National Vulnerability Database. (2010). Retrieved from https://nvd.nist.gov/

[11]: Bob Martin et.al. (September 2011). 2011 CWE/SANS Top 25 Most Dangerous Software Errors. Retrieved from http://cwe.mitre.org/top25/archive/2011/2011\_cwe\_sans\_top25.pdf

[12]: Rik Farrow. (n.d.). Foundations: What Are Buffer Overflows?

Retrieved from http://www.watchguard.com/infocenter/editorial/135136.asp

[13]: Thomas Schwarz. (n.d.). Buffer-overflow Attack. Retrieved from http://www.cse.scu.edu/~tschwarz/coen152\_05/Lectures/BufferOverflow.html

[14]: Desheng Fu and Feiyue Shi. (November, 2012). Buffer Overflow Exploit and Defensive Techniques. Retrieved from http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=6405636&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs\_all.jsp%3Farnumber%3D6405636

[15]: Buffer-overflow diagram. (n.d.). Retrieved from http://cis1.towson.edu/~cssecinj/wp-content/uploads/cs2/buffer-overflow.jpg

[16]: Kevin Snow. (n.d.). Memory Exploits and Defenses. Retrieved from http://cs.unc.edu/~fabian/courses/CS600.624/slides/exploits.pdf

[17]: Matthew R. Fletcher. (2005). Simulating Hardware-level Buffer-Overflow Protection.

ProQuest Information and Learning Company, Ann Arbor, MI

[18]: Wenliang Du. (2006). Return-to-libc Attack Lab. Retrieved from http://www.cis.syr.edu/~wedu/seed/Labs/Vulnerability/Return\_to\_libc/Return\_to\_libc.pdf

[19]: Takahiro Shinagawa. (October 2006). SegmentShield: Exploiting Segmentation Hardware for Protecting against Buffer Overflow Attacks. Retrieved from http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4032489

[20]: H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. (October,

2004). On the effectiveness of address-space randomization. In *Proc. 11th ACM Conference on Computer and* *Communications Security* (*CCS ’04*) , pages 298–307.

[21]: Sirisara Chiamwongpaet and Krerk Piromsopa. (2009). The Implementation of Secure Canary Word for Buffer-Overflow Protection. Retrieved from http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=5189584&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs\_all.jsp%3Farnumber%3D5189584

[22]: K. Piromsopa. (November, 2006). Secure Bit: Transparent, Hardware Buffer-Overflow Protection.Retrieved from http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=4012648&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs\_all.jsp%3Farnumber%3D4012648