Buffer Overflows and Available Countermeasures

1. Introduction:

With the increasing use of internet, the opportunities for attempts to access remote systems illegally have been increased using various techniques and methods. Some attacks merely annoy or occupy system resources for example system attacks by virus and Trojans. However, other attacks are more pernicious because they seize root privileges and modify, corrupt, or steal data. Today most common form of security vulnerability is the buffer overflow and this is most serious class of security threat because buffer overflow vulnerabilities give the attacker ability to inject and execute attack code.

Vulnerabilities that could lead to buffer overflow attack are still in a significant proportion in languages like C and C++ and these vulnerabilities allow attackers to inject malicious codes as these languages does not automatically bounds-check array and pointer references and more importantly, many of the functions provided by the standard C library are unsafe.

In an article of CERT Larry Rogers states that Buffer overflows have been a problem in software-based systems and applications for a long time. One of the first significant computer break-ins that took advantage of a buffer overflow was the Morris worm, and that happened in November 1988. Fingerd service, a service that dispenses information about the set of users logged into UNIX-based computer systems was affected by this worm. Since 1988 buffer overflow has accounted for large fraction of internet based attacks. For example, in 2001, according to CERT advisory CA-2001-19 “code red” worm exploited buffer overflow in Microsoft IIS indexing service DLL and it affected more than 250,000 hosts and after that in 2003 SQL slammer worm attacked machines running Microsoft SQL server 2000.

Figure 1 shows that there is increase in number of CERT security advisories in the previous years that were based on buffer overflow.
There are various vulnerability exists in computer applications because all application work on the stack and heap buffers. Sometimes for exploiting the application, attacker writes the past boundaries of a stack buffer and overwrites the return address of the function. However, several other vulnerabilities also exist that allow an attacker to inject code into an application. Such vulnerabilities can also occur when dealing with dynamically allocated memory by heap.

Since no return addresses are available in the heap, an attacker must overwrite other data stored in the heap to inject code. The attacker could overwrite a pointer located in this memory, but since these are not always available in heap allocated memory, attackers often overwrite the management information that the memory allocator stores with heap-allocated data (Younan, Joosen, and Piessens, 2006). Heap buffer overflow is described later in more detail.

This essay describes what buffer overflow is and effective countermeasures available for mitigation. For example Secure Designing and Coding, Configuring non-executable stack, Use of safer versions of functions, Use of safe libraries, tools for detection and prevention of buffer overflow, Canary-based defences, Adding bounds to all buffers and protection of return address to being overwritten etc.

Next section will explain buffer overflow in more detail. After that, section 3 describes the exploitation of buffer overflow and available countermeasures are discussed in section 4.
2. **Buffer overflow**

A buffer overflow is an anomaly wherein the data transferred to a buffer overruns the original storage capacity of the buffer and some of the data "overflows" into neighbouring buffer, one that the data was not intended to go into.

Programming language like C and C++ provide no built in protection against the accessing or overwriting the data in any part of the memory and the capability of automatic bound checking. For example strcat(), strcpy(), sprintf(), vsprintf(), bcopy(), gets() and scanf() calls in c language can be exploited as the function quit testing if any buffer in the stack is not as large as data copied into the buffer. Therefore programmers have to explicitly check the use of these functions cannot overflow buffers. However, they often omit these checks. Consequently, many programs are plagued with buffer overflows and are therefore vulnerable to security attacks.

The following C programme illustrates an example of simple buffer overflow:

```c
#include<stdio.h>
int main(int argc, char **argv)
{
    char abc[5] = “lanc”;
    char xyz[10] = “Lancaster”;
    strcpy( xyz, “Lancasteruniversity”);
    printf(“%n”, abc);
    return 0;
}
```

As the strncpy function of C programme copies the 19 characters into the xyz buffer whose memory space is only 11 characters, as there is no space for the remaining characters it eats up the memory space of abc buffer, destroying the contents of the abc buffer. Due to this the output of this programme in memory looks like this:

```
\0\0l a n c
\0\0c a s t e
\0\0r u n i v
```

However, the output should be following in the memory stack if the buffer overflow did not happen:

```
\0 l a n c
\0 l a n c
a s t e r
```

Since buffers can only hold a specific amount of data, when that capacity has been reached the data has to flow somewhere else, typically into another buffer, which can
Buffer overflow and available countermeasures

Corrupt data already in that buffer and result in erratic program behaviour, including memory access errors, a crash, incorrect results, or a breach of system security. (Piromsopa and Enbody, 2006)

Sometime this overflow injects additional code into an unsuspecting process and then hijacks control of that process to execute the injected code. The hijacking of control is usually accomplished by overwriting return addresses on the process stack or by overwriting function pointers in the process memory. In either case, an instruction that alters the control flow (such as a call, return, or jump instruction) may inadvertently transfer execution to the wrong address that points at the injected code instead of the intended code (Baratloo, Singh, and Tsai, 2000).

2.1. Types of buffer overflows:

Stack based buffer overflow

Stack-based buffer overflows have been considered the common type of exploitable programming errors found in the software applications. Stack overflow occurs when program writes more data to the buffer located on the stack than the actual buffer size of the stack. If the stack buffer is supplied with the data that comes from the untrusted user then in that case stack buffer can be exploited by injecting executable code into running program and take control over it. Hackers mostly rely on stack based overflow flow because of the stack’s vulnerable nature.

Heap based buffer overflow:

A heap overflow is a type of buffer overflow that occurs in the heap data area and a heap is an area of memory utilized by an application which is allocated dynamically at runtime. Unlike stack overflow heap buffer overflow can be inconsistence and more complex but heap corruption can be more powerful. In the heap buffer an application allocates memory dynamically as needed and this allocation occur through function call malloc(). Heap based buffer overflows are not so common because it is very difficult the guess the memory address which is dynamically allocated.

3. Buffer overflow exploits:

The most common form of security attacks can be accomplished by achieving two goals first injecting the attack code, which can be sequence of instructions that exploit the running process and secondly, changing the execution path of running process to execute the attack code. It is important to note that these two goals are interrelated to each other because injecting a code in process and not using it doesn’t make any sense. The most popular form of buffer overflow exploits are known as stack smashing attack: modifying return address saved on the stack and heap smashing attacks: exploitation of buffer
overflow in dynamic allocated memory. Some new techniques are also developed like arc injection and pointer subterfuge based upon stack and heap based overruns.

The arc injection technique (sometimes referred to as return-into-libc) involves a control transfer to code that already exists in the program’s memory space. Various pointer subterfuge techniques change the program’s control flow by attacking function pointers (pointer variables whose value is used as an address in a function call) as an alternative to the saved return address, or modify arbitrary memory locations to create more complex exploits by subverting data pointers. (Pincus and Baker, 2004, p.20)

Stack smashing attack:

Stack smashing technique is based upon the fact most C compilers store the saved return address on the same stack used for local variables. In addition to this, for stack smashing attack, an attacker have to supply payload (combination of code and data to achieve a particular goal). This payload value can be supplied during the input process of programme or by sending a network packet to a well known port of a system. Following example illustrate the stack smaching attack.

```c
void fun(void* arg, size_t len) {
    char buff[100];
    memcpy(buff, arg, len); /* buffer overrun if len > 100 */

    /* .. fun body */
    return;
}
```

In the function called fun, if the arg value contains some executable code and the program counter is modified to point to the address of buff. When the memcpy function will return, the system will execute the code that is inserted at buff, instead of returning to the calling procedure.

Arc injection:

Modification of the control flow of programme is done in this technique instead of injecting executable code. An example below illustrates overview of arc injection technique in a stack.
Buffer overflow and available countermeasures

1. Normal stack
2. Stack when attacker calls a function
3. Stack smashing

Figure: 2

Above figure shows that when there is no bound checking in the functions it is easy for an attacker to exploit the stack segment by inserting large chunk of data which may overwrite the stack segment and corrupt the return address of the function call. This corruption help to insert the new return address which alters the control flow programme.

Arc injection techniques are especially useful in the case where programme being attacked has some sort of memory protection for example nonexecutable stacks. In this technique no attacker supplied code is executed in the stack so mitigation techniques like nonexecutable stack do not prevent arc injection exploits.

Pointer subterfuge:

Pointer subterfuge is a general term for exploits that involve modifying a pointer’s value (Pincus and Baker, 2004, p.23). there are four different types of pointer subterfuge exist: function pointer clobbering, data-pointer modification, and virtual pointer smashing.

1. Function pointer clobbering: modification of function pointer to point to the attacker supplied code. In this type of attack when a function pointer is used to execute a programme call, attacker code is executed instead of intended code. This type of attack is very effective in the situation where the program uses some form of mitigation techniques (StackGard) for the prevention of saved return address. As
the saved return address is not overwritten so this mitigation cannot protect such attacks.

2. Data pointer modification: in this attacker takes control over an address which is used as a target for subsequent assignment and this control can let him/her to modify other memory locations for attack. Data pointer modification is used for more complex exploits as it can be combined with function pointer clobbering and traditional stack smashing techniques. This technique is efficient in situation where more straightforward buffer overflow techniques are used.

3. Virtual pointer smashing:

Replacing an object’s virtual pointer to an attacker supplied virtual function table (VTBL) is known as virtual pointer smashing. Such attack allows the attacker to transfer control when the next virtual function is invoked. Virtual pointer attack can be implemented on the programs based upon the C++ language because most C++ compiler implement virtual function via virtual function table (VTBL) associated with each class.

VPTR smashing attacks can apply both to stack buffer overruns and heap buffer overruns, and can be used in conjunction with code injection or arc injection (Pincus and Baker, 2004, p.25)

Heap smashing attack:

As discusses so far traditional approach called stack smashing is more popular and vulnerable but now a days various patches are implemented for the protection from such attacks so attackers are using heap smashing approach for exploiting. Though, heap smashing techniques are complicated as attacker typically does not know the heap block’s location ahead of time and in many cases it is difficult to predict when heap free operation will happen. Moreover, in some situations it is difficult to predict whether the next block has been allocated at the time of overflow. Surprisingly, now a day’s attackers are becoming more advanced and they are exploiting the dynamic memory allocation on which heap buffers are based.

function pointer clobbering, data-pointer modification, and virtual pointer smashing all these can be used for exploiting heaps as well. Figure 3, 4 and 5 explains the concept of heap overflow attack.
Buffer overflow and available countermeasures

Figure 3

If the virtual table is on the heap next to a string object then heap may look like this:

Figure 4

After buffer overflow:

Figure 5

This shows that when heap is exploited by buffer overflow it can alter the virtual function table and when next time virtual function is invoked it will execute the attacker’s code or attacker’s virtual table for exploitation.
4. Available countermeasures:

There are various tools and techniques available for preventing and mitigation of buffer overflow attacks but this essay concentrate only on specific techniques like compiler modification/programming language, use of safe libraries, pointer protection, executable space protection and address-space randomization. Stack protection is based on the two mechanisms: canary based and non-canary based. Canary based technique is used for protecting return address of the stack and it is known for excellent performance but this does not prevent all buffer overflow attacks. However, non-canary based gives more protection but require performance overhead. Before going in deep it is important to understand the concept of canary.

Canary value works for the protection of stack buffer overflow, a known value called canary is inserted onto the runtime stack below the return address of executing function. Canary value has been checked by the stack protection mechanism after function execution is finished. To find out progress of buffer overflow attack, modification in the canary value is checked. If there is any modification happened in the canary value protection mechanism stop the execution and throws an error message. There are three popular canary based protection available known as StackGard, Stack Smashing Protector and libverify.

1. Canary based:

1.1 StackGuard:

StackGuard is a compiler extension which is based on the canary value checks. A randomly chosen canary value is inserted in the stack immediately below the return address on function entry to protect the return address. StackGuard is implemented by stack smashing protector, which is a GCC( Gnu compiler collection) extension for protecting application from stack smashing attacks. After applying these extensions programs are safe from buffer overflow attacks.

StackGuard prevents changes to active return addresses by either detecting the change of the return address before the function returns, or by completely preventing the write to the return address. Detecting changes to the return address is a more efficient and portable technique, while preventing the change is more secure. StackGuard supports both techniques, as well as adaptively switching from one mode to the other (Cowan, 1988).

1.2 Libverify:

Libverify is the dynamically loadable libraries that implement return address verification similar to the StackGard but libverify does not require recompilation of source code. Special libraries contain all code and canary values for verification. In contrast to StackGuard, which generates random numbers for use as canaries, libverify uses the actual return address as the canary value for each function (Baratloo, Singh, and Tsai, 2000). Furthermore, libverify injects the verification code
at the start of the process execution via binary re-writing of the process memory while StackGuard uses canary verification at compile time.

For verification, linking of the libraries to the user code is necessary and this includes certain steps. At first init() function in library is executed which contain code to instrument the process such that canary verification code in the library will be called for all functions in user code. There are three main steps defined by (Baratloo, Singh, and Tsai, 2000) for instrumentation.

1. Determine the location and size of the user code.
2. Determine the starting addresses of all functions in the user code.
3. for each function
   (a) Copy the function to heap memory.
   (b) Overwrite the first instruction of the original function with a jump to the wrapper_entry function.
   (c) Overwrite the return instruction of the copied function with a jump to the wrapper_exit function.

After the instrumentation canary value is saved on the canary stack by wrapper_entry function and the wrapper_exit function verifies the current canary value to the saved canary stack value. If the values does not match then the function determines that buffer overflow is occurred and stops the programme further execution.

2. Non canary based:

2.1 Use of safe language

Programs written in C and C++ languages are more vulnerable to buffer overflow attacks because these languages do not follow the automatic bound checking mechanism. Languages like LISP, Java, Ada, small talk, and c derivatives as cyclone and D eliminates the possibility of buffer overflow attacks as these languages perform additional checks at runtime.

2.2 Libsafe

Libsafe is safe library implementation which is dynamically loadable and it intercepts all calls made to C standard library function known to be vulnerable to buffer overflow attacks. (Baratloo, Singh, and Tsai, 2000) says The key idea behind libsafe is the ability to estimate a safe upper limit on the size of buffers automatically. In this technique calculation of buffer size is made after the start of the function in which the buffer is accessed.

Libsafe method is able to determine the maximum buffer size by realizing that local buffers cannot extend beyond the end of the current stack frame. This realization allows the substitute version of the function to limit buffer writes within the estimated
buffer size. Thus, the return address from that function, which is located on the stack, cannot be overwritten, and control of the process cannot be commandeered (Baratloo, Singh, and Tsai, 2000). Though use of libsafe require little performance overhead but this technique is most effective in prevention of buffer overflow attacks.

2.3 Pointer protection by PointGuard:

Attacks which are based upon pointers like code pointer attack and data pointer attack are more dangerous and desirable as the attack seeks total control of victim process and attacker can get to a root privileged shell. It is important to seek out some effective defence mechanism for preventing programs from such kind of attacks so a technique called PointGuard was presented by (Cowan, Beattie, Johansen and Wagle, 2003) to defend against most kind of buffer overflow attacks by encrypting pointers when stored in memory, and decrypting them only when loaded into CPU registers.

Encryption technique to protect pointers used by PointGuard is very effective because it is not possible for attacker to corrupt pointer in a way that decryption of corrupt pointer gives them their desired result. PointGuard technique is critically depends on strategy that pointer always being loaded into registers prior to being dereferenced. This technique is used for making the pointers safe in registers as registers are not addressable via computed addresses and that’s why they are not subject to overflow attacks. The attack always fails if PointGuard technique is applied for preventing a program from buffer overflow attack because when the attacker’s corrupted value is passed through the PointGuard decryption process then it will produce a random address reference which would not be desired by the attacker for exploit. In addition to this, PointGuard provides integrity and confidentiality for pointers by making it very difficult for attacker to predict the encryption key or look at cypher text because a new key is always generated at the time the process starts.

Furthermore, PointGuard’s overhead is low when protecting real security-sensitive applications such as OpenSSL, and show that PointGuard is effective in defending against buffer overflow vulnerabilities that are not blocked by previous defences (Cowan, Beattie, Johansen and Wagle, 2003).

2.4 Executable space protection:

Executable space protection is an approach in which execution of malicious code is prevented by making the stack or heap address area nonexecutable. There are patches available that can make the stack or heap address spaces nonexecutable and they offer an advantage of zero performance penalty. Moreover, programs are protected without recompilation but they need these special patches for protection. Executable space protection technique is not an efficient protection against all buffer overflow attacks because of its disadvantages. First of all in order to make a
nonexecutable space it is necessary to patch the operating system kernel which is not possible in all closed source operating system like windows. In addition to this, if the attack does not rely on code execution then this protection mechanism is not successful. Some Unix based operating systems for example OpenBSD and Mac OS X use this approach for buffer overflow protection and now new variant of Microsoft also support executable space protection.

2.5 Address space randomization:

This is also a powerful technique for preventing some buffer overflow attacks by making it difficult for an attacker to predict target addresses. For example to execute an injected code on the stack has to find the stack first and if memory addresses are randomized then it will be difficult to guess the address.

Randomizing the address-space layout of a software program prevents attackers from using the same exploit code effectively against all instantiations of the program containing the same flaw (Shacham, Page, and Pfaf, 2004).

5. Conclusion:

Despite the diverse nature of all the potential solution given above, no silver bullet is available for solving all buffer overflow vulnerabilities. All techniques are limited in one manner to another. Moreover, attackers have a long history of learning how to beat detection and prevention mechanisms so the best possible solution for the buffer overflow attacks is to write the secure error free code and minimize the use of standard unsafe functions. In addition to this, warning given by compilers should be considered seriously by programmers. Furthermore, most of the vulnerabilities are inadvertently introduced in applications because of the lack of training in secure coding practices among programmers and sometime vulnerabilities exists in application because of the tight delivery schedules and other more persistence requirements. Vulnerability related to buffer overflow are so complex that some time it is good to introduce additional protection mechanism like address space randomization and use of save libraries for effective prevention and mitigation from buffer overflow attacks.
References


