**Breesy Reyes**
**Sam Chen**

# Assignment 6 - Buffer Overflow

**1. Define buffer overflow.**
Buffer overflow is when you have an interface that allows you to input more data into a buffer or data holding area than the amount allocated which can overwrite other information. This can be exploited if the attacker crashes the system or inserts code that allows them to gain control to the system.

**2. What types of programming languages are vulnerable to buffer overflows?**
Assembly (machine) languages and languages such as C and its derivatives because they provide the ability to access and manipulate memory data.

**3. What are the two broad categories of defenses against buffer overflows?**
The two categories of defenses against buffer overflow are compile-time defenses or run-time defences. Compile-time defenses aim to prevent or detect overflows by instrumenting programs when they are compiled. Run-time defenses aim to provide some protection for existing vulnerable programs.

**4. Describe how a return-to-system-call attack is implemented and why it is used.**
This attack uses stack overflow. It replaces the return address of an overflowing stack. Normally, a stack is filled with instructions to execute but return to system call replaces the return address of a stack by another instruction. This is used because it allows attackers to access functions in a program without putting in any malicious code. It can be used to avoid the non executable stack limitation.

**5. Describe how a heap buffer overflow attack is implemented.**
Attackers will use the input of applications by overflowing the buffer memory. The three locations in the address space that are vulnerable are the stack, heap, and data section. The heap management pointers are targeted, then the memory has been allocated and included in executing the program data, finally the data in the heap gets corrupted to make the application overwrite pointers in the heap.

**6. Describe how a global data area overflow attack is implemented.**

The targeted area is the global data area. The buffer used here has been overwritten by using unsafe buffer operations therefore, changing the pointer memory locations used in the buffer. The program thats being attacks will make a call to this overwritten function which will transfer control to the shellcode to the attacker.

**7. Rewrite the program shown below so that it is no longer vulnerable to a buffer overflow.**

```
int main ( int argc , char * argv []) {
        int valid = FALSE ;
        char str1 [8];
        char str2 [8];
        next_tag(str1);
        //we use fget() because it guarantees to not read over one less
        than the buffers size of characters
        fgets (str2, sizeof(str2), stdin);
        if (strncmp ( str1 , str2 , 8) == 0)
                valid = TRUE ;
        printf (“buffer1 : str1 (%s), str2 (%s), valid (%d)\n”, str1 , str2 , valid) ;
}
```

**8. Rewrite the function shown below so that it is no longer vulnerable to a stack buffer overflow.**

```
 void gctinp (char * inp, int siz)
{
        puts (“Input value : “);
        fgets (inp, siz, stdin);
        printf (“buffer3 getinp read %s\n”, inp) ;
}
void display (char * val)
{
        char tmp [16];
        //we use snprintf() it guarantees that not too much data is
        written into the buffer.
        snprintf (tmp, sizeof(temp), “read val : %s\n”, val) ;
        puts (tmp) ;
}
```

```
        int main ( int argc , char * argv [])
        {
                char buf [16];
                getinp (buf, sizeof(buf)) ;
                display (buf) ;
                printf ("buffer3 done \n") ;
         }
```

**---------- different way**

```
Void getinput(char *inp, int size){
        puts("Enter input");
        fgets(inp, size, stdin);
        printf("got input");
        if(inp[strlen(inp)-1] != '\n'){
                Extra = 0;
                while(((ch = getchar() != '\n') && (ch != EOF))
                        Extra = 1;
        }
        if(extra == 1){
                printf("Too long to fit. Try again.);
        }
}
Void display(char *inp){
        Char temp[16];
        printf(temp, "read value: %s\n", inp);
        puts(temp);
        }
Int main(){
        Char buff[16];
        getinput(buff,sizeof(buf));
        display(buf);
        printf("Buffer Done");
}
```

**9. Rewrite the two functions shown below so they are no longer vulnerable to a buffer overflow**
**attack.**

**int copy_buf ( char * to , int pos , char * from , int len )**

```
        {
                int i;
                for ( i =0; i < len ; i ++) {
                        to [ pos ] = from [ i ];
                        pos ++;
                }
                return pos ;
        }
        short read_chunk ( FILE fil , char * to )
        {
                short len ;
                fread (& len , 2 , 1 , fil ) ;
                fread ( to , 1 , len , fil ) ;
                return len ;
        }
---------------------------------------------------- safe code
Int copy_buff(char *to, int size, int pos, char *from, int len){
        Int i;
        if(len<=0)
                Return pos;
        if((pos+len) > size)
                len=size - pos;
        for(i=0;i<len;i++)
                to[pos]=from[i];
                Pos++;
        Return pos;
}


Short read_chunk(FILE fil, int size, char *to){
        Short len;
        fread(&len,2, 1, fil);
        if(len<=0)
                Return 0;
        if(len>size)
                Return size;
        fread(to,1,len,fil);
```

```
        Return len;
}
```

**10. Rewrite the program shown below so that it is no longer vulnerable to a heap buffer overflow.**

```
        /* record type to allocate on heap */
        typedef struct chunk {
                char inp [64]; /* vulnerable input buffer */
                void (* process ) ( char *) ; /* pointer to function to process inp */
        } chunk_t ;
        void showlen ( char * buf )
        {
                int len ;
                len = strlen ( buf ) ;
                printf (" buffer5 read %d chars \n", len ) ;
        }
        int main ( int argc , char * argv [])
        {
                chunk_t * next ;
                setbuf ( stdin , NULL ) ;
                next = malloc ( sizeof ( chunk_t ) ) ;
                next - > process = showlen () ;
                printf (" Enter value : ") ;
                fgets(next->inp, SIZE, stdin); //SAFE CODE
                //gets ( next - > inp ) ;//DELETE
                next - > process ( next - > inp ) ;
                printf (" buffer5 done \n") ;
        }
```