

## *Malicious Code Insertion Example*

This code has a bug that allows a hacker to take control of its execution and run evilfunc().

```
#include <stdio.h>

// obviously it is compiler dependent
// but on my system this function is
// located at address x300E (12302 decimal)
void evilfunc()
{
    printf("EVIL!  EVIL!\n");
}

int getsum()
{
    int nums[3];
    {
        // this block ensures that the nums
        // array is the first local variable
        // in the stackframe.

        int i = 0;
        int sum = 0;
        int input = 0;

        printf("Enter up to 3 integers.  Type 0 to quit.\n");

        /* scanf gets an integer that the user enters */
        scanf("%d", &input);

        while(input != 0) {
            nums[i] = input;
            printf("Entered %d\n", nums[i]);
            sum = sum + nums[i];
            i++;
            scanf("%d", &input);
        }
        return sum;
    }
}

int main()
{
    int result = getsum();
    printf("Sum is %d", result);
    return 0;
}
```

## Mike's IO Functions Cheat Sheet

### Regular IO Functions

FILE \*fopen(const char \*FILE, const char \*MODE) – open a file. Check “man fopen” for modes.

int fclose(FILE \*fp) – close an open file. Zero return means no error.

int putchar(int CH) – write a character to stdout

int putc(int CH, FILE \*FP) – write a character to the file

int getchar() – read a character from stdin. Returns EOF if at the end of the file (check for this).

int getc(FILE \*FP) – read a character from the file. Returns EOF if at the end of the file (check for this).

char \*fgets(char \*s, int size, FILE \*stream) - reads in at most one less than size characters from stream and stores them into the buffer pointed to by s. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A '\0' is stored after the last character in the buffer. This is usually the preferred method for reading in string data.

### Formatted IO Functions

<pre>int printf(const char *format, ...); int fprintf(FILE *stream, const char *format, ...); int sprintf(char *str, const char *format, ...);</pre>	Print formatted output to stdout/a file/a string respectively. Check “man 3 printf” for details.
<pre>int scanf(const char *format, ...); int fscanf(FILE *stream, const char *format, ...); int sscanf(const char *str, const char *format, ...);</pre>	Reads data from stdin/a file/a string respectively. Has built in conversions for things like integers and floats. Returns the number of successfully matched items. Check “man 3 scanf” for details.

### Example

```
#include <stdio.h>

int main()
{
    /* open testfile.txt for reading and appending */
    char *filename;
    FILE *myFile;
    char inputBuffer[10];
    int intOne;

    filename = "testfile.txt";
    myFile = fopen(filename, "a+");
    if(myFile == NULL) {
        printf("Could not open %s\n", filename);
        return(1);
    }
    /* read up to 9 characters from the user, or to a newline */
    fgets(inputBuffer, 10, stdin);
    /* write what was read to the file */
    if(fprintf(myFile, "%s", inputBuffer) < 0) {
        /* error handling. We'll close the file and quit. */
        fclose(myFile);
        return(1);
    }
    /* read what we wrote to the file or to a newline*/
    fgets(inputBuffer, 10, myFile);
    printf("Read %s from file\n", inputBuffer);
    /* try to parse an int out of the string we read */
    if(sscanf(inputBuffer, "%d", &intOne) == 1) {
        /* we know an integer was successfully read */
        printf("Read int %d\n", intOne);
    }
    fclose(myFile);
    return 0;
}
```

## Mike's Varargs Cheat Sheet

### *How to Write Your Varargs Function*

1. `#include <stdarg.h>`
2. Decide how you want to know when you have processed all the arguments. You can pass along a parameter that tells you how many arguments to expect (either just a int or a fancy string like `printf` uses). You can also have a special “stopping value” like `NULL` that tells you you're done.
3. Write your header like “`int myfunc(int var1, char* var2,...)`”
4. Declare a `va_list` variable: `va_list myargs;`
5. Pass that variable to `va_start` along with the last “normal” parameter of your function:  
`va_start(myargs, var2);`
6. As long as you still need more parameters, you use `va_arg` and pass along the parameters type:  
`myParameterInt = va_arg(myargs, int);`
7. When you know you're finished you call `va_end`: `va_end(myargs);`

### *Example*

```
/* prints any number of string parameters on different lines */
void printOnLines(int numberOfItems, ...)
{
    va_list args;
    int i;
    va_start(args, numberOfItems);
    for(i = 0; i < numberOfItems; i++) {
        char * currentString = va_arg(args, char *);
        printf("%s\n", currentString);
    }
    va_end(args);
}

int main() {
    printOnLines(3, "Hello", "World", "Varargs");
    return 0;
}
```

## Mike's Malloc Realloc & Free Cheat Sheet

`void *malloc(size_t size)` - allocates `size` bytes and returns a pointer to the allocated memory. If not enough memory is available, `malloc` will return `null`. You always must check for this.

`void free(void *ptr)` - frees memory that was allocated with `malloc` (or `realloc`). If the pointer you pass was not allocated with `malloc` or has already been freed once, `free` may corrupt memory in unpredictable ways.

`void *realloc(void *ptr, size_t size)` - allocates a new chunk of memory and copies everything that exists in the old location to the new one (this allows you to in essence “extend” a chunk of memory you already malloced). Returns a pointer to the new memory location, and frees the old memory location. If not enough memory is available, `realloc` will return `null` but the old memory will not be freed. You must always check for this.

### Sample code:

```
#include <stdlib.h>

/* dynamically allocates an array of ints */
int mallocSample(int n)
{
    int i;
    int *ip;
    int *tmp;
    /* first we allocate n ints */
    if((ip = malloc(n*sizeof(*ip))) == NULL) {
        /* Handle Error Here */
    }
    /* we initialize all those ints to zero */
    for(i = 0; i < n; i++)
        ip[i] = 0;

    /* if we discover we need a little more
       space we can use realloc. Lets expand to
       10 more ints than we did before */

    if((tmp = malloc((n+10)*sizeof(*ip))) == NULL) {
        /* Handle Error Here. Bear in mind that ip
           is not freed if there is an error. So if
           we want to free it we must say... */
        free(ip);
        /* more error code here */
    } else {
        /* tmp now points to our new data. ip has
           been freed. So we can say... */
        ip = tmp;
    }

    /* do some more stuff with ip, maybe initialize those
       10 new values */

    /* eventually we must free the memory we've allocated
       or its a memory leak. */
    free(ip);
    return 0;
}
```

## Kitten Explosion Exercise

1. \_\_\_\_\_

2. \_\_\_\_\_

3. \_\_\_\_\_

4. \_\_\_\_\_

5. \_\_\_\_\_

6. \_\_\_\_\_

7. \_\_\_\_\_

8. \_\_\_\_\_

9. \_\_\_\_\_

10. \_\_\_\_\_

11. \_\_\_\_\_

12. \_\_\_\_\_

13. \_\_\_\_\_

14. \_\_\_\_\_

15. \_\_\_\_\_

16. \_\_\_\_\_