Integer Overflow Lab

Introduction

In this laboratory we are going to look at some examples of issues that can cause vulnerabilities in applications. Specifically we are going to introduce integer overflows.

Integer Overflow

Program code 1:

```c
#include<stdio.h>
int main(){
        unsigned char n=250;
        int i;

        for(i=0;i<10;i++){

                printf("%hhu | %hhX \n", n, n);
                n++;
        }
        printf("n is %hhu and n-10 equals %hhu\n", n,n-10);
}
```

Program code 2:

```c
#include<stdio.h>
int main(){
         char i;

         for(i=0;i<128;i++){

                  printf("%hhd | %hhX \n", i, i);

         }
}
```

Program code 3:

```c
#include<stdio.h>
int main(){
        int i=2;

        while(i>0){
                i=i*i;
                printf("%d \n", i);
        }
}
```

Program code 4:

```c
#include<stdio.h>

int main(){
    char i=0;
    int a;
    char b[150];
    for(a=0;a<150;a++){
        b[a]='a';
    }
    b[149]='\0';
    while(i<150){
        printf("i is %hhd\n", i);
        printf("%c\n",b[i]);
        i++;
    }

}
```

1) Copy the aforementioned codes and save them into individual files (e.g. intergeroverflow.c). Identify what the programs do.

2) Compile them using the following command:

```
gcc -o integeroverflow integeroverflow.c -w -g
```

The “-w” flag inhibits issues that are noted by the compiler (there shouldn’t be any for this programme, but we will use it later - you should NEVER compile code with the “-w” flag set in the future). The “-g” flag provides some extra help when investigating the executable file in the debugging/disassembling software that we are going to use - gdb.

3) Run the programmes and observe the outputs.

4) Open the executable files in the gdb environment using:

```
gdb integeroverflow
```

Once in the environment we can “list” the program code, set execution breakpoints, run the code and look at the contents of the registers using the following:

```
list
break x //where x is a line number
run //run the code up to the break point
continue //continue to execute the code after the break point
info register //show the contents of the registers
x/x &i //show the memory contents for the variable i
```

5) Using gdb look at the memory to identify what is happening when an integer overflows/wraps around.

Variable Scope

Copy the following code to a C file; compile it and run the program to appreciate the scope of various variable types.

```c
#include<stdio.h>

int j=1000;


void fun3(){
        int i=7;
        int j = 10;
        printf("\t\t\t[in fun3] i @ 0x%08x =%d\n", &i, i);
        printf("\t\t\t[in fun3] j @ 0x%08x =%d\n", &j, j);
}


void fun2(){
        int i=5;

        printf("\t\t[in fun2] i @ 0x%08x =%d\n", &i, i);
        printf("\t\t[in fun2] j @ 0x%08x =%d\n", &j, j);
        j=1337;
        printf("set j = 1337\n");
        fun3();
        printf("\t\t[back in fun2] i @ 0x%08x =%d\n", &i, i);
        printf("\t\t[back in fun2] j @ 0x%08x =%d\n", &j, j);

}


void fun1(){
        int i=3;

        printf("\t[in fun1] i @ 0x%08x =%d\n", &i, i);
        printf("\t[in fun1] j @ 0x%08x =%d\n", &j, j);
        fun2();
        printf("\t[back in fun1] i @ 0x%08x =%d\n", &i, i);
        printf("\t[back in fun1] j @ 0x%08x =%d\n", &j, j);
}

int main(){
        int i=1;
        printf("[in main] i @ 0x%08x =%d\n", &i, i);
        printf("[in main] j @ 0x%08x =%d\n", &j, j);
        fun1();
        printf("[back in main] i @ 0x%08x =%d\n", &i, i);
        printf("[back in main] j @ 0x%08x =%d\n", &j, j);
        return 0;

}
```

Memory Segmentation

Copy the following code to a C file; compile it and run the program to appreciate different parts of memory.

```c
#include<stdio.h>
int global_var;
int global_initialized_var=5;

void function(){
        int stack_var;
        printf("The memory location of function stack_var is @ 0x%x\n", &stack_var);

}
int main(){
        int stack_var;
        static int static_initialized_var=5;
        static int static_var;
        int *heap_var_ptr;

        heap_var_ptr = (int *)malloc(4);
        //in the date segment
        printf("global_initialized_var is at address 0x%x\n", &global_initialized_var);
        printf("static_initialized_var is at address 0x%x\n", &static_initialized_var);
        //in the bss segment
        printf("global_var is at address 0x%x\n", &global_var);
        printf("static_var is at address 0x%x\n", &static_var);
        //in the heap segment
        printf("heap variable is at address 0x%08x\n", heap_var_ptr);
        //in the stack segment
        printf("stack variable is at address 0x%08x\n", &stack_var);

        function();

        return 0;

}
```

Additional task

Copy the following code to a C file; compile it and run the program. Try various inputs and see if you can discover any vulnerability and exploit the program (i.e. print the “The ‘modified’ variable is changed…..” on the command line).

Hint (gdb is your friend)

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

int main(int argc, char **argv){
        volatile int modified;
        char buffer[32];
        modified=0;
        strcpy(buffer, argv[1]);

        if(modified!=0){
                printf("The 'modified' varialbe is changed. Well done!\n");

        }else{
                printf("Try again?\n");
        }
        return 0;

}
```