

CS392/CS681 - Computer Security

Lab 7 - Robust Programming

Due November 12/01/03

Introduction: Fragile, or non-robust, programs are major causes for information security flaws. A program is considered fragile when it terminates unexpectedly or produces unexpected results, when used improperly. A robust program, on the other hand, anticipates such improper usage and terminates gracefully without causing any damage to the system's security. In part I of this lab you will be applying principles and guidelines you studied in the class to harden a program. The techniques used in part I are effective in protecting against bad programmer, but they often don't protect against end users who use compiled programs. A typical example is buffer overflow attacks as you have seen the previous lab. In part II of this lab you will create an attack tree; a systematic way of representing approaches to breach a system with specific objectives in mind. In part III of this lab you will learn to harden your FEAU utility against these attacks such as buffer overflow attacks and format string attacks by creating a robust API for developers.

Pre-requisites:

- You must read "*Robust Programming*" by Matt Bishop to do this LAB
- Its highly suggested you look at <http://www.counterpane.com/attacktrees.pdf> by Bruce Schneier (Counterpane Systems)
- Read the article "<http://www-106.ibm.com/developerworks/security/library/s-buffer-defend.html> " and paper "<http://www.mcs.csuhayward.edu/~simon/security/boflo.html> "

The Task:

Part I: Consider the following binary tree library

bintree.h:

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

struct tree_el {
    int val;
    struct tree_el * right, * left;
};

typedef struct tree_el node;
void insert(node ** tree, node * item);
void printout(node * tree);
void add_int(node ** tree, int value);
void del_tree(node * root);
```

bintree.c:

```
#include "bintree.h"
void insert(node ** tree, node * item) {
    if(!(*tree)) {
```

```

*tree = item;
return;
}

if(item->val<(*tree)->val)
    insert(&(*tree)->left, item);
else if(item->val>(*tree)->val)
    insert(&(*tree)->right, item);
}

void printout(node * tree) {
    if(tree->left) printout(tree->left);
    printf("%d\n", tree->val);
    if(tree->right) printout(tree->right);
}

void add_int(node ** tree, int value){
    node * curr;
    curr = (node *)malloc(sizeof(node));
    curr->left = curr->right = NULL;
    curr->val = value;
    insert(&(*tree), curr);
}

void del_tree (node * root) {
    if(!root)
        return;
    del_tree(root->left);
    del_tree(root->right);
    free(root);
}

```

The above library could be considered fragile library code. It works perfectly fine as long as the user who uses it knows what s/he is doing. Let's say a user has written the following code:

```

#include "bintree.h"

void main() {
    node * root;
    unsigned long int i;
    root = NULL;
    i=0;
    do{
        add_int(&root, rand());
        i++;
    }while(i>0);

    printout(root);
    del_tree(root);
}

```

Study the code closely and start thinking of how you can re-write bintree library to prevent such a disaster. The above code will compile and execute without any errors, but will consume enough resources to render your system useless (unless, of course, you restart your machine or find and kill the process). This is only an example of what a

naïve programmer can do. There are millions of other ways a programmer can write a program that can be harmful, so as a robust programmer it is your responsibility to write robust libraries that can anticipate multiple odd situations and handle them appropriately.

Part I: Re-write bintree to handle at least five situations presented in “Robust Programming”.

Part II: Create an Attack Tree to illustrate the means to achieving several goals of breaching the security of FEAU. Those goals are:

- Breaking the Authentication: For example, fooling the system you are someone else
- Breaking the Authorization: For example, bypassing or cracking the password procedure
- Breaking the Privacy / Confidentiality: For example, bypassing or cracking the encryption mechanisms
- Breaking the Integrity: For example, bypassing or cracking the hashing mechanisms

On your leaf nodes indicate (where applicable):

1. The feasibility of the attack: “Possible” by a (p) and improbable by a (i). This should be obvious to you but if it isn’t the other metrics below may shed light on it.
2. The time complexity of the attack: “Time Consuming” by a (T) and non-time consuming as (Q) for quick. What is “Time Consuming” is left up to you.
3. The requirement of special equipment: “Special Equipment” by a (SE) and “No Special Equipment” as (NSE). The definition of special equipment is left up to you but for example brute force AES requires a lot of processors and time

Part III: Read the articles related to buffer overflows and find all possible buffer overflow vulnerabilities in FEAU utility, report them, and you must show how to exploit the at least two vulnerabilities.

Post LAB report: Your report must include following:

- List of situations you have anticipated
- Modified and well commented bintree.h, bintree.c and a demonstration program to show that your code is robust
- A summary of “Robust Programming”
- A report on your implemented and tested system
- Soft and hard copy of your implementation
- The Attack Tree that illustrates the possible breaks in the security principles and policies
- List all possible buffer overflow or format string vulnerabilities you found in FEAU utility that lead to the leaf nodes of the Attack tree (the actual attacks), and describe how a malicious user would exploit them

Send question and submission to fscarimbolo@nyc.rr.com.