Secure C Programming: Memory Management
Coursework 02

Introduction

This coursework introduces students to basic approaches to specify and verify security vulnerabilities in C programs considering memory safety aspects. In particular, this coursework provides theoretical and practical exercises to *(i)* identify and describe software vulnerabilities concerning memory safety in C programs; *(ii)* apply software model checking to detect such vulnerabilities automatically; *(iii)* analyze the counterexample produced by typical software model checkers; and lastly *(iv)* describe how to fix the software vulnerabilities identified by (bounded) model checking techniques based on the diagnostic counterexample.

Learning Objectives

By the end of this lab you will be able to:

- Understand risk assessment to guide software developers.
- Review dynamic data structures.
- Provide rules for secure coding in the C programming language.
- Develop safe, reliable, and secure C software.
- Eliminate undefined behaviours that can lead to exploitable vulnerabilities.

1) **(Risk Assessment)** Identify the vulnerabilities and indicate the potential consequences of not addressing them in the following fragments of C code.

i. Variable-length automatic arrays.

```c
int foo(int n, int b[], int size) {
    int a[n], i;
    for (i = 0; i < size + 1; i++) {
        a[i] = b[i];
    }
        return i;
}

int main() {
        int i, b[100];
        for (i = 0; i < 100; i++) {
                b[i] = foo(i, b, i);
        }
        for (i = 0; i < 100; i) {
                if (b[i] != i) {
                        ERROR:  return 1;
                }
        }
        return 0;
}
```

ii. Dynamic memory allocation.

```
#include <stdlib.h>
int *a, *b;
int n;
#define BLOCK_SIZE 128
void foo () {
  int i;
  for (i = 0; i < n; i++)
    a[i] = -1;
  for (i = 0; i < BLOCK_SIZE - 1; i++)
    b[i] = -1;
}
int main () {
  n = BLOCK_SIZE;
  a = malloc (n * sizeof(*a));
  b = malloc (n * sizeof(*b));
  *b++ = 0;
  foo ();
  if (b[-1])
  { free(a); free(b); }
  else
  { free(a); free(b); }
  return 0;
}
```

iii. Linked list implementation.

```
#include <stdlib.h>
void myexit(int s) {
        _EXIT: goto _EXIT;
}
typedef struct node {
  int h;
  struct node *n;
} *List;
int main() {
  /* Build a list of the form 1->...->1->0 */
  List a = (List) malloc(sizeof(struct node));
  if (a == 0) myexit(1);
  List t;
  List p = a;
  a->h = 2;
  while (__VERIFIER_nondet_int()) {
    p->h = 1;
    t = (List) malloc(sizeof(struct node));
    if (t == 0) myexit(1);
    p->n = t;
    p = p->n;
  }
  p->h = 2;
  p->n = 0;
  p = a;
  while (p!=0) {
    if (p->h != 2) {
      ERROR: __VERIFIER_error();
    }
    p = p->n;
  }
  return 0;
}

```

2) **(Diagnostic Counterexamples)** Bounded model checking (BMC) is an automatic verification technique for checking software systems. The basic idea of BMC is to check the negation of a given property at a given depth: given a transition system M, a property φ, and a bound k, BMC unrolls the system k times and translates it into a verification condition (VC) ψ such that ψ is satisfiable if and only if φ has a counterexample of depth k or less. Here, the tasks consist of:

i. Verify the C programs from question 1) using the ESBMC model checker (http://esbmc.org/). You should explore the different options for property checking (e.g., pointer safety, memory leak, bounds check) and verification strategies (e.g., falsification and incremental BMC) available in ESBMC.
ii. Identify the root cause of the vulnerabilities identified in the C programs of question 1) based on the counterexample produced by ESBMC.
iii. Fix the vulnerabilities that were identified in *ii* by analyzing the diagnostics counterexamples.

References:

[1] Dirk Beyer: *Automatic Verification of C and Java Programs: SV-COMP 2019*. TACAS (3) 2019: 133-155.

[2] Mikhail Y. R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, Denis A. Nicole: *ESBMC 5.0: an industrial-strength C model checker*. ASE 2018: 888-891.