

Secure C Programming: Memory Management

Coursework 02

Introduction

This coursework introduces students to basic approaches to specify, verify, and understand 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 techniques to detect such vulnerabilities automatically; (iii) analyze the counterexample produced by state-of-the-art software model checkers; and lastly (iv) describe how to fix the software vulnerabilities identified by software 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 security vulnerabilities and indicate the potential consequences of not addressing them via risk assessment. You must consider the following fragments of C code to answer this question, which were extracted from the International Competition on Software Verification (SV-COMP) [1].

i. Variable-length automatic arrays.

```
1 int foo(int n, int b[], int size) {
2     int a[n], i;
3     for (i = 0; i < size + 1; i++) {
4         a[i] = b[i];
5     }
6     return i;
7 }
8
9 int main() {
10     int i, b[100];
11     for (i = 0; i < 100; i++) {
12         b[i] = foo(i, b, i);
13     }
14     for (i = 0; i < 100; i) {
15         if (b[i] != i) {
16             ERROR: return 1;
17         }
18     }
19     return 0;
20 }
```

ii. Dynamic memory allocation.

```
1 #include <stdlib.h>
2 int *a, *b;
3 int n;
4 #define BLOCK_SIZE 128
5 void foo () {
6     int i;
7     for (i = 0; i < n; i++)
8         a[i] = -1;
9     for (i = 0; i < BLOCK_SIZE - 1; i++)
10        b[i] = -1;
11 }
12 int main () {
13     n = BLOCK_SIZE;
14     a = malloc (n * sizeof(*a));
15     b = malloc (n * sizeof(*b));
16     *b++ = 0;
17     foo ();
18     if (b[-1])
19     { free(a); free(b); }
20     else
21     { free(a); free(b); }
22     return 0;
23 }
```

iii. Linked list implementation.

```
1 #include <stdlib.h>
2 void myexit(int s) {
3     _EXIT: goto _EXIT;
4 }
5 typedef struct node {
6     int h;
7     struct node *n;
8 } *List;
9 int main() {
10     /* Build a list of the form 1->...->1->0 */
11     List a = (List) malloc(sizeof(struct node));
12     if (a == 0) myexit(1);
13     List t;
14     List p = a;
15     a->h = 2;
16     while (__VERIFIER_nondet_int()) {
17         p->h = 1;
18         t = (List) malloc(sizeof(struct node));
19         if (t == 0) myexit(1);
20         p->n = t;
21         p = p->n;
22     }
```

```

23  p->h = 2;
24  p->n = 0;
25  p = a;
26  while (p!=0) {
27      if (p->h != 2) {
28          ERROR: __VERIFIER_error();
29      }
30      p = p->n;
31  }
32  return 0;
33 }
34

```

2) **(Diagnostic Counterexamples)** Bounded model checking (BMC) for software is an automatic verification technique for checking security properties in software systems [2]. 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 software 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, you must accomplish the following tasks:

- i. Verify the C programs from question 1) using the ESBMC model checker (<http://esbmc.org/>) [3]. Note that ESBMC can also be used via a web interface available at <http://18.225.0.33/esbmc.php>. You must 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 security vulnerabilities identified in the C programs of question 1) based on the counterexample produced by ESBMC. Here you must be able to reproduce the security vulnerabilities by following the sequence of steps provided in the counterexample.
- iii. Fix the vulnerabilities that were identified in item (ii) by analyzing the diagnostics counterexamples and then verify the fixed version with ESBMC to make sure that you have fixed the vulnerabilities.

3) **(Memory Alignment in C)** Identify in which fragments of C code the data structures are not aligned considering a 32-bit architecture [4]. Once you identify these issues with data structure alignment, you must add padding bytes to ensure a proper alignment of its members.

- i. `MixedData1` consists of four members.

```

struct MixedData1
{
    char a;
    short int b;
    int c;
    char d;
};

```

ii. MixedData2 consists of three members.

```
struct MixedData2
{
    short s;
    int i;
    char c;
};
```

iii. MixedData3 consists of three members.

```
struct MixedData3
{
    int i;
    char c;
    short s;
};
```

References:

- [1] Dirk Beyer: *Automatic Verification of C and Java Programs: SV-COMP 2019*. TACAS (3) 2019: 133-155.
- [2] Lucas C. Cordeiro, Bernd Fischer, João Marques-Silva: *SMT-Based Bounded Model Checking for Embedded ANSI-C Software*. IEEE Trans. Software Eng. 38(4): 957-974 (2012).
- [3] 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.
- [4] MSDN article on data alignment. [https://docs.microsoft.com/en-us/previous-versions/ms253949\(v=vs.90\)?redirectedfrom=MSDN](https://docs.microsoft.com/en-us/previous-versions/ms253949(v=vs.90)?redirectedfrom=MSDN).