A Static Analysis Method for Detecting Buffer Overflow Vulnerabilities

Testing programs for vulnerabilities

Zhang Shuhao, Qu Shaobo
Qin Jianxing, Hong Yun

Mentor: Hugh Anderson, Harish Venkatesan

2022.7.28

Overview

Introduction

Buffer overflow

Symbolic execution overview

Symbolic Execution

Demo

GUI display

Overview

Introduction

Buffer overflow

Symbolic execution overview

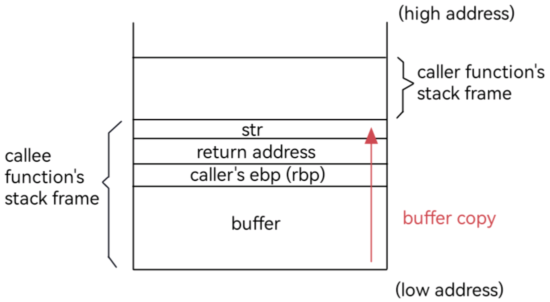
Symbolic Execution

Demo

GUI display

Definition

Occurs when data > fixed-length block of memory

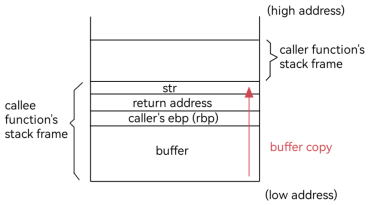


- commonly seen
- easy to be unaware of
- can be fatal

Exploitation

The techniques to exploit a buffer overflow vulnerability vary by architecture, by operating system and by memory region.

Take stack buffer overflow as an example:



- use extra data to overwrite return address
- point to a malicious program

Overview

Introduction

- Buffer overflow
- Symbolic execution overview

Symbolic Execution

- Demo
- GUI display

Overview

Symbolic execution is a way of executing a program abstractly

- assume symbolic values for inputs

$$a = 1, b = a + 2$$
$$\Rightarrow a = \alpha, b = \alpha + \beta$$

- focuses on execution paths

Good points

- multiple possible inputs can be covered by doing one execution
- avoid reporting false warnings

Overview

Introduction

- Buffer overflow
- Symbolic execution overview

Symbolic Execution

- Demo
- GUI display

Demo

Here is a piece of C code:

```
void foo(int *arr, int n, int h) {
    ASSUME(n > 0, capacity(arr) >= n);
    int idx;
    if (h != 0) {
        idx = 0;
    } else {
        idx = n;
    }
    print(arr[idx]);
}
```

`ASSUME` is a annotation that represents constraints on the inputs.

A programmer familiar with C may see that `print(arr[idx])` can cause buffer overflow when the capacity of `arr` is exactly `n` and it also happens that `h` is 0.

Let's see how symbolic execution can find this vulnerability.

Initialization

```
void foo(int *arr, int n, int h) {
    ASSUME(n > 0, capacity(arr) >= n);
    int idx;
    if (h != 0) {
        idx = 0;
    } else {
        idx = n;
    }
    print(arr[idx]);
}
```

$$E = \{arr \mapsto \mathtt{ptr}(0, arr_{cap}), n \mapsto n', h \mapsto h'\}$$

$$P = n' > 0 \land arr_{cap} \geq n'$$

Declaration

```
void foo(int *arr, int n, int h) {
    ASSUME(n > 0, capacity(arr) >= n);
    int idx;
    if (h != 0) {
        idx = 0;
    } else {
        idx = n;
    }
    print(arr[idx]);
}
```

$$E = \{arr \mapsto \mathtt{ptr}(0, arr_{cap}), n \mapsto n', h \mapsto h', idx \mapsto idx'\}$$

$$P = n' > 0 \land arr_{cap} \geq n'$$

Branching and assignment

```
void foo(int *arr, int n, int h) {
    ASSUME(n > 0, capacity(arr) >= n);
    int idx;
    if (h != 0) {
        idx = 0;
    } else {
        idx = n;
    }
    print(arr[idx]);
}
```

$$E_1 = \{arr \mapsto \mathtt{ptr}(0, arr_{cap}), n \mapsto n', h \mapsto h', idx \mapsto 0\}$$

$$P_1 = n' > 0 \wedge arr_{cap} \geq n' \wedge h' \neq 0$$

$$E_2 = \{arr \mapsto \mathtt{ptr}(0, arr_{cap}), n \mapsto n', h \mapsto h', idx \mapsto n'\}$$

$$P_2 = n' > 0 \wedge arr_{cap} \geq n' \wedge h' = 0$$

Branch merging

```
void foo(int *arr, int n, int h) {
    ASSUME(n > 0, capacity(arr) >= n);
    int idx;
    if (h != 0) {
        idx = 0;
    } else {
        idx = n;
    }
    print(arr[idx]);
}
```

$$E_1 = \{arr \mapsto \mathtt{ptr}(0, arr_{cap}), n \mapsto n', h \mapsto h', idx \mapsto 0\}$$

$$E_2 = \{arr \mapsto \mathtt{ptr}(0, arr_{cap}), n \mapsto n', h \mapsto h', idx \mapsto n'\}$$

$$E = \{arr \mapsto \mathtt{ptr}(0, arr_{cap}), n \mapsto n', h \mapsto h', idx \mapsto \mathtt{if}\ h' \neq 0\ \mathtt{then}\ 0\ \mathtt{else}\ n'\}$$

$$P = n' > 0 \wedge arr_{cap} \geq n'$$

Memory access

```
void foo(int *arr, int n, int h) {
    ASSUME(n > 0, capacity(arr) >= n);
    int idx;
    if (h != 0) {
        idx = 0;
    } else {
        idx = n;
    }
    print(arr[idx]);
}
```

$$E = \{arr \mapsto \mathtt{ptr}(0, arr_{cap}), n \mapsto n', h \mapsto h', idx \mapsto \mathtt{if}\ h' \neq 0\ \mathtt{then}\ 0\ \mathtt{else}\ n'\}$$

$$P = n' > 0 \wedge arr_{cap} \geq n'$$

$$C = P \wedge ((\mathtt{if}\ h' \neq 0\ \mathtt{then}\ 0\ \mathtt{else}\ n') < 0 \vee (\mathtt{if}\ h' \neq 0\ \mathtt{then}\ 0\ \mathtt{else}\ n') \geq arr_{cap})$$

C is satisfiable! Consider $n' = arr_{cap} = 1, h' = 0$

Satisfiability solving

z3 is a SMT solver, which can "magically" solve the final problem.

You may use its Python binding like this:

```
import z3

n = z3.Int('n')
h = z3.Int('h')
idx = z3.If(h!=0, 0, n)
cap = z3.Int('cap')
s = z3.Solver()
s.add(z3.And(n>0, cap>=n, z3.Or(idx<0, idx>=cap)))
if s.check() == z3.sat:
    print(s.model())
```

It will outputs a model `[h = 0, n = 1, cap = 1]`.

Overview

Introduction

- Buffer overflow
- Symbolic execution overview

Symbolic Execution

- Demo
- GUI display