Inter-procedural buffer overflows detection in C/C++ source code via static analysis

Irina Dudina

ИСП РАН

Institute for System
Programming of the Russian
Academy of Sciences

Lomonosov Moscow State
University

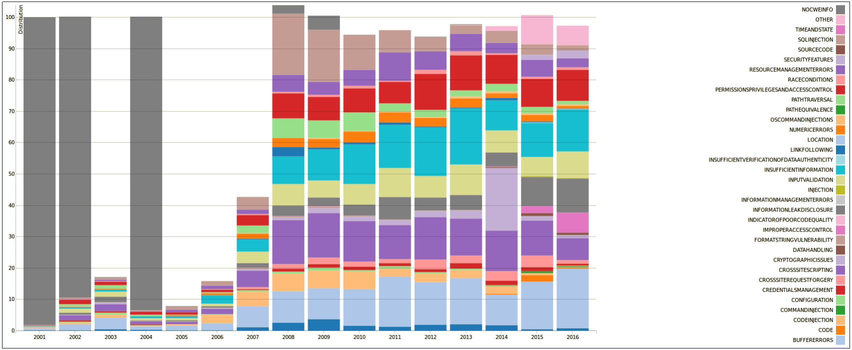
Moscow
1 dec 2016.

Buffer overflow error

ИСП РАН

Accessing (reading or writing) the buffer with value that exceedes it's bounds.

- may lead to program fall, incorrect operation of the program, security vulnerability;
- for the last ten years remains one of the most common source of vulnerability.



https://nvd.nist.gov/visualizations/cwe-over-time

Buffer overflow detection

Detecting buffer overflow vulnerabilities by analyzing code in general is an undecidable problem.

> The halting problem can be trivially reduced to the buffer overflow detection problem.

Analyzer use cases define the treshold between:

- recall,
- scalability,
- resource consumption,
- speed,
- FP rate.

Goal

Developing buffer overflow detector within **SVACE** .

Requirements

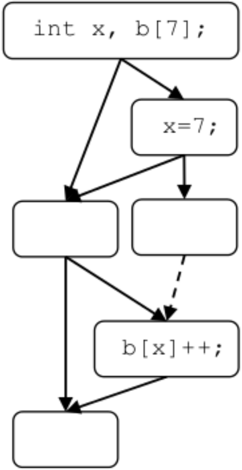
- hight scalability,
  full **Android** analysis in 5 hours (all detectors),
- hight True Positive rate (50% – 70%),
- *path-sensitivity*,
- *inter-procedural detection*,
- warning message with detailed trace.

Limitatons

- Buffers on stack or on static memory with compile-time known size only.

Approach based on critical edge detection

A BOF warning is fired if CFG of a function contains an edge, for which every path containing this edge always has out of bounds buffer access.



Advantages

- Hight TP rate.

  This BOF defect definition is based on the assumption that programmers do not write unreachable code, which is satisfied for the vast majority of cases.
- Efficient detection algorithm.

Flaw

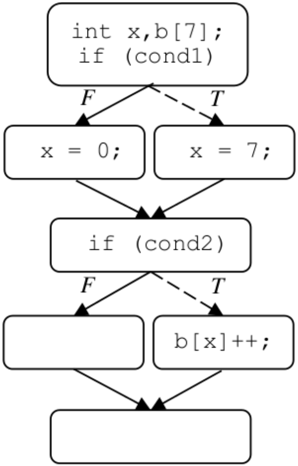
- Missing real defects.

  Not all BOF errors satisfy this criteria.

Defect depending on path conditions

Out of bounds access happens if `cond1` && `cond1` is satisfiable.

- No critical edge.
- Satisfiability of conjunctions of path conditions must be checked to report such kind of an error.



Unknown function precondition

```c
#define S 10
int buf[S];
```

```c
void foo(int idx) {
  buf[idx]++;
}
```

```c
int bar(int a, int b) {
  if (a >= S-1) {
    // ...
  }
  if (b)
    a++;
  return buf[a];
}
```

BOF if `idx >= S`

BOF if `a >= S - 1 && (b)`

Unknown function precondition

```
#define S 10
int buf[S];
```

```
void foo(int idx) {
  buf[idx]++;
}
```

```
int bar(int a, int b) {
  if (a >= S-1) {  //true
    // ...
  }
  if (b)    //true
    a++;
  return buf[a];
}
```

BOF if `idx >= S`

For the single path BOF depends on `idx`

BOF if `a >= S - 1 && (b)`

The path contains BOF regardless input values

Unknown function precondition

```
#define S 10
int buf[S];
```

```
void foo(int idx) {
  buf[idx]++;  //OK
}
```

```
int bar(int a, int b) {
  if (a >= S-1) {  //true
    // ...
  }
  if (b)    //true
    a++;
  return buf[a];  //BOF here
}
```

BOF if `idx >= S`

For the single path BOF depends on `idx`

BOF if `a >= S - 1 && (b)`

The path contains BOF regardless input values

Formal definition of BOF

A function is said to have a BOF defect if it's CFG contains a path, for which the following hold

- it contains an access to the buffer of size `S` with index `i`,
- for each set of input values either this path is infeasible or `i` $\notin$ [0, `S`-1],
- it is feasible for at least one set of input values*.

*We don't have any information about possible sets of input values according to the precondition. Hence, in case where this faulty path is forbidden by the precondition we will have a FP-warning.

SVACE infrastructure

SVACE core is responsible for base analyses, such as building CFG, unreachable code detection, detecting functions terminating program, etc.

SVACE core performs value numbering which produces a set of value identificators (VId).

Each detector can use results of core analyses and it operates by mapping some properties to value indentificators ($v \in VId$) at every program point $q \in Instr$. These mappings are known as *attributes*.

$$Attr : VId \times Instr \to AttrVal$$

SVACE core performs symbolic execution with state merging. It notifies all detectors about all program events. To develop a checker one should specify handlers for essential events.

During symbolic execution **SVACE** computes necessary reachability conditions for all program points as formulas on value identificators.

$$ReachCond : Instr \to Cond$$

$NotLess$ and $NotGreated$ formulas

Suppose for particular $q \in Instr$ and $v \in VId$ and arbirtrary $x \in VId$ we have formula $NotLess(q, v, x)$.

*For arbirtrary x, $NotLess(q, v, x)$ is a suffisient condition for the fact that if execution reached q, than it went along a path, on which **always** (regardless function input values) $v \geqslant x$.*

Similarly for $NotGreater(q, v, x)$.

Then, for the point $ac \in Instr$, where buffer of size S is accessed by $i \in VId$, a suffisient condition of overflow (according to the definition) is satisfiability of

$$ReachCond(q) \wedge (NotLess(ac, i, v_s) \vee NotGreater(ac, i, v_{-1})),$$

where v_s and v_{-1} are value identificators for constants S and -1 respectively.

Example

Why do we need such complex condition?

```
 1 #define S 10
 2 int buf[S];
 3 int foo(int a,
 4         int c)
 5 {
 6   int idx;
 7   if (c > 7)
 8     idx = 10;
 9   else
10     idx = a;
11   if (c < 15)
12     return buf[idx];
13   return a;
14 }
```

$$
\begin{aligned}
NotLess(p_{12}, idx, x) &= (c > 7) \land (10 \geq x)\\
NotGreater(p_{12}, idx, x) &= (c > 7) \land (10 \leq x)\\
ReachCond(p_{12}) &= (c < 15) \land\\
&((c > 7 \land (idx = 10)) \lor (c \leq 7 \land (idx = a)))\\
BOF\_Cond &= (c < 15) \land (c > 7)\\
&\land (idx = 10) \land (10 \geq S)
\end{aligned}
$$

Satisfiable ($a = 0, c = 8, idx = 10$)
$\Rightarrow$ fire warning! (faulty path 6-7-8-11-12)

Just $idx \geq S$ will not work because of the unknown precondition.
Eg precondition is ($a \leq 9$), but as long as we don't know it, we will report warning for $a = 42, c = 3, idx = 42$.

Calculating $NotLess$ and $NotGreated$ formulas

Calculating these formulas for constants is trivial.

$$Eg. \qquad NotLess(q, c, x) = (c \geq x)$$

If you know these formulas for a and b you can calculate formulas for

- values that were compared to a in some dominant vertex (eg. for t if in current point holds $t > a$),
- the result of an arithmetic operation $r = a \diamond b$,
- join value of a and b in the merged state (eg. `c = cond ? a : b`),
- ...

Inter-procedural BOF detection

Three points on the program form BOF:

- index definition,
- buffer definition,
- access instruction.

All three poins can belong to the different functions, so the warning should be reported in the closest common ancestor (in cycle-free callgraph).

SVACE uses function summaries to perform inteprocedural analysis. All functions are analyzed from bottom to top in the callgraph and results of the callee analysis are used in the caller.

Inter-procedural calculation of $NotLess$ and $NotGreater$ formulas

```c
#define S 10
int buf[S];
int foo(int a,
        int c)
{
  int idx;
  if (c > 7)
    idx = 10;
  else
    idx = a;
  if (c < 15)
    return buf[idx];
  return a;
}
```

- Propagating whole formula for variables which are present in caller to the caller context when applying the summary.
- Creating stubs for formal argument while calculating formula. They will be replaces with formula for actual argument in caller context if it has one.

```c
r = foo(k, c);
```

$$NotLess(k) = P$$

$$NotLess(r) = (c \geq 15) \wedge (r = k) \wedge P$$

Conditional facts of buffer access inside a function

```c
#define S 10
int buf[S];
int foo(int a,
        int c)
{
  int idx;
  if (c > 7)
    idx = 10;
  else
    idx = a;
  if (c < 15)
    return buf[idx];
  return a;
}
```

- access to the buffer with known size,

```
access to buffer of size: S
              with index: a
     if (c <= 7) && (c < 15)
```

- access to the buffer, passed as an argument.

Implementation

BUFFER_OVERFLOW.EX – main checker.
BUFFER_OVERFLOW.LIB.EX – uncorrect use of libcalls.
OVEFLOW_AFTER_CHECK.EX – BOF in cycle.

Evaluation results on Android 5.0.2

Warning type	Count	TP, %
BUFFER_OVERFLOW.EX	221	62
BUFFER_OVERFLOW.LIB.EX	64	64
OVERFLOW_AFTER_CHECK.EX	66	67