

boa



Buffer Overrun Analyzer

Edo Cohen
039374814
sedoc@t2

Tzafrir Rehan
039811880
tzafrir@cs

Gai Shaked
036567055
gai@tx

March 1, 2011

Chapter 1

Introduction

1.1 Goal

Given a C program that performs buffer manipulations, statically (at compile time) identify whether the program may perform array access out of the array bounds.

1.2 Previous work

Chapter 2

boa

2.1 Overview

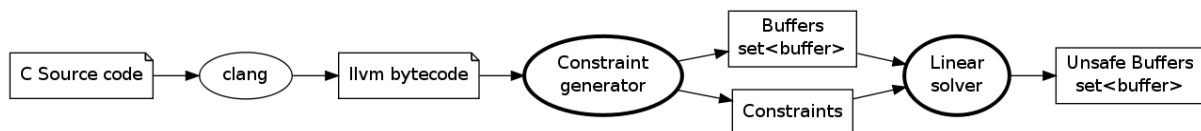


Figure 2.1: Main components and stages

2.2 Constraint Generator

2.2.1 Integers

2.2.2 Direct array access

```
1 char buf[10];
2 buf[10] = 'a';
```

2.2.3 String manipulation functions

```
1 #include "string.h"
2
3 int main() {
4     char *str1 = "longer_than_ten", *str2 = "short";
5     char buf1[10], buf2[10];
6     strcpy(buf1, str1);
7     strcpy(buf2, str2);
8 }
```

2.2.4 Buffer aliasing

2.3 Linear Solver

The constraints generated represent a linear problem, and each solution of the problem suggests a set of ranges for the values each integer may receive and the allocation and usage of each buffer. As we aim to find the tightest ranges, therefore we direct our linear solver to find a solution maximizing -

$$Goal = \sum_{\text{Buffers}} \left[\{\text{buf!used!min}\} + \{\text{buf!alloc!min}\} - \{\text{buf!used!max}\} - \{\text{buf!alloc!max}\} \right]$$

A solution satisfying this goal will maximize the lower bounds and minimize the upper bounds of each buffer access, and thus assure we get the tightest solution.

Once we have the solution¹ we test each buffer to verify that -

$$\{\text{buf!used}\} \subseteq \{\text{buf!alloc}\}$$

Which means -

$$\begin{aligned} \{\text{buf!used!max}\} &< \{\text{buf!alloc!min}\} \\ \{\text{buf!used!min}\} &\geq 0 \end{aligned}$$

Note that we stick to the size and numbering conventions of C, safe access to a buffer of size n is any access to the cells $0 \dots n - 1$. If the solution does not satisfy one of the constraints, we report a possible buffer overrun in this specific buffer.

2.3.1 Elastic filter

In many cases, the constraints we generate create an infeasible linear problem. The simplest example of such case is -

```
1  int i;
2  i++;
```

The constraint generated from the second line will be -

$$\begin{aligned} \{\text{tmp!max}\} &\geq \{\text{i!max}\} + 1 \\ \{\text{i!max}\} &\geq \{\text{tmp!max}\} \end{aligned}$$

Which is obviously an infeasible set of constraints. The same problem holds in many different cases, including *strcat* (which concat one string to the end of another, and therefore imply equivalent set of constraints to the string used length).

When our linear solver discover that the constraints problem we have generated is infeasible, we wish make the smallest change to the problem and make it feasible once again. There is a great body of work in the area of finding and eliminating IIS (*irreducibly inconsistent system*), the common and naive

2.3.2 Blame system

2.4 Implementation

¹The solution is a set of integer value, one for each of the problem variables, such that all the constraints are satisfied and the *Goal* value is maximized

Chapter 3

-To be named-

3.1 Test system

3.2 Version control

3.2.1 Code reviews

Chapter 4

Results

We tested *boa* on several widespread real world programs. We tested to see whether *boa* discover real buffer overruns, and also to evaluate the number of false alarms and their main causes. The source files used in all of the experiments is available in *boa* git repository[2].

Table 4.1 summarize the performance of *boa* on the programs we present in this document, the reported running times are the results of experiments on Dell vostro 1310 laptop, with Intel Core2 Duo CPU T8100 2.10GHz and 2GB RAM running Debian GNU/Linux Wheezy (7.0.0), clang 2.9, llvm 2.9 and GLPK 4.43. On this humble configuration *boa* can analyze few thousands lines of code within seconds, thus the use of elastic filter did pay off and *boa* can be used to efficiently analyze any reasonable piece of C code.

	fingerd	flex	syslog
Source lines	230		332
Constraints	2894		1206
Running time	2.508s		1.304s
Buffers	34		15
Overruns reported	6		8
Real overruns	1		1

Table 4.1: *boa* performance on various real world examples

4.1 fingerd

We tested *boa* using *fingerd*, unix finger daemon. We altered the current source code to reflect the well known buffer overrun, used by the Internet worm in 1988. The overrun is caused by using the unsafe function *gets* to read data into the 1024¹ bytes buffer *line*. As far as we know, this is the only real buffer overrun in the 230 lines source code.

Running on *fingerd* source, *boa* report overruns on 6 out of the 34 buffers. Next we present *boa*'s blame three of them, and analyze the reason for the reported overrun -

line is the only real overrun in *fingerd*

```
line tests/realworld/fingerd/fingerd.c:85
- unsafe function call gets [tests/realworld/fingerd/fingerd.c:121]
- unknown function call realhostname_sa [tests/realworld/fingerd/fingerd.c:128]
- memchr call might read beyond the buffer [tests/realworld/fingerd/fingerd.c:139]
[ ... 10 more lines ... ]
```

The overrun discovered by *boa*, and the real cause reported briefly. Note another result of *gets* - every other buffer access based on *line*'s length will be reported as an overrun.

¹Back in 1988 *line* was 512 bytes, but it does not matter for the analysis.

rhost is a char buffer ment to hold the host name

```
rhost tests/realworld/fingerd/fingerd.c:86
- unknown function call realhostname_sa [tests/realworld/fingerd/fingerd.c:128]
- unknown function call realhostname_sa [tests/realworld/fingerd/fingerd.c:128]
- buffer alias with offset [tests/realworld/fingerd/fingerd.c:128]
- buffer alias with offset [tests/realworld/fingerd/fingerd.c:128]
```

This false alarm is caused by the use of *realhostname_sa*, from *socket.h*. This false alarm could be avoided if boa would model *socket.h* functions, but even now the output let the user identify the cause immediatly and decide manually wether this call is safe or not.

malloc is a generic name for any buffer created by a malloc call, one can distinguish between two malloc calls by thies source location (filename and line number)

```
malloc tests/realworld/fingerd/fingerd.c:141
- buffer alias [tests/realworld/fingerd/fingerd.c:149]
- buffer alias [tests/realworld/fingerd/fingerd.c:149]
- buffer alias [tests/realworld/fingerd/fingerd.c:149]
- buffer alias [tests/realworld/fingerd/fingerd.c:141]
- buffer alias [tests/realworld/fingerd/fingerd.c:149]
```

This blame might seem wierd at a first look, how comes buffer alias alone cause an overrun? But the solution appears quickly by looking at the source lines (141, 149) reffered by the blame -

```
141      if ((t = malloc(sizeof(line) + 1)) == NULL)
```

...

```
149      for (end = t; *end; end++)
150          if (*end == '\n' || *end == '\r')
151              *end = '_';
```

The programmer allocate a buffer large enough to include *line*, and then iterates through the array using the ++ operator on a pointer. Since boa is a flow-insensitive analyzing tool, we can not assure that the incremental pointer aliasing will be limited to the buffer size - and therefore boa report a possible buffer overrun.

4.2 flex

Bibliography

- [1] Buffer Overrun Detection using Linear Programming and Static Analysis
- [2] boa git repository - <https://github.com/tzafrir/boa/>
- [3] Locating Minimal Infeasible Constraint Sets in Linear Programs