# boa


Buffer Overrun Analyzer


| Edo Cohen | Tzafrir Rehan | Gai Shaked |
| --- | --- | --- |
| 039374814 | 039811880 | 036567055 |
| sedoc@t2 | tzafrir@cs | gai@tx |

Project in advanced programming - program analysis (236503)
Computer science department, Technion - Israel institute of technology

# Chapter 1

# Introduction

## 1.1 Goal

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

# Chapter 2

# boa

## 2.1 Overview

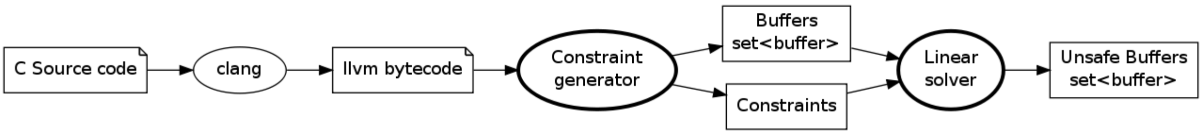


Figure 2.1: Main components and stages

## 2.2 Constraint Generator

### 2.2.1 Integers

### 2.2.2 Direct array access

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

### 2.2.3 String manipulation functions

```
#include "string.h"

int main() {
  char *str1 = "longer␣than␣ten", *str2 = "short";
  char buf1[10], buf2[10];
  strcpy(buf1, str1);
  strcpy(buf2, str2);
}
```

### 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 recieve and the allocation and usage of each buffer. As we aim to find the tightest ranges, we direct our linear solver to find a solution maximizing -

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

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 solution[1] 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 Handling infeasible problems

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

```
int i;
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 concatenates one string to the end of another, and therefore implies an equivalent set of constraints to the string used length).

When our linear solver discovers 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*), and we follow the algorithms and terminology of Chinneck and Dravnieks[3]. The common and naive approach is the deletion filtering -

1. input: $Q$ is an infeasible set of constraints
2. [try to delete] FOREACH $q_i \in Q$ DO:
   (a) Test whether $Q \backslash q_i$ is feasible -
      i. IF infeasible - set $Q = Q \backslash q_i$
3. $Q$ is an IIS

After one iteration the algorithm return an IIS, which can be removed from the original problem. In case there are several IISs in the original problem - the algorithm should repeat until the problem become feasible. We have implemented this approch at first, and it did work well on small pieces of code, but naturally did not scale well - on the same testing system described in chapter 4 it took more than half an hour to eliminate the IISs in the 400 lines of source of *md5* library, and more than 8 hours to find the blames (section 2.4) as well. Therefore we read furthr and implemented an elastic filter for eliminating IIS.

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

#### 2.3.1.1 Elastic filter

The main idea behind elastic filtering is adding a new *elastic variable* to each constraint, allowing it to *strech* and therefore the infeasibility removed, than we solve the new problem, trying to minimize effect of the *elastic variables* and each elastic variable assigned a non-zero value marks a constraint which is part on an IIS. Formally -

1. Initialize $S = \emptyset$ (will hold the IIS)
2. $Q$ is an infeasible set of constraints of the form -
$$q_i : \sum_{j=1}^{n_i} a_{i_j} X_{i_j} \geq c_i$$
where $a_{i_j}, c_i$ are constants and $X_{i_j}$ is a variable of the constraint problem.
3. Add an *elastic variable* $e_i$ to each constreaint $q_i \in Q$ such that - $q_i : \sum_{j=1}^{n_i} a_{i_j} X_{i_j} + e_i \geq c_i$
4. Limit the elastic variables to accept non-negative values, and set the goal of the linear problem to minimize the sum of the elastic variables - $Goal = -\sum e_i$
5. WHILE the problem is feasible -
   (a) Solve the linear problem, for each elastic variable $e_i$ -
      i. If $e_i > 0$ -
         A. $S = S \cup \{q_i\}$
         B. remove $e_i$ ($q_i : \sum_{j=1}^{n_i} a_{i_j} X_{i_j} \geq c_i$)
6. return $S$

The most time consuming part of the process is trying to solve the constraint problem (the linear solver use simplex). The delete filter tries to solve $n$ problems ($n$ - the number of constraints in the original problem) and return one IIS. On the other hand - the elastic filter perform only $m$ iterations ($m$ - the number of constraints in the smaller IIS in the problem), and return $m$ constraints of each of the IISs in the problem. A typical IIS in the problems generated by our constraint generator consist of 3 or 4 constraints, thus we have predicted the performence of the elastic filter will be mush better than the deleting filter, and indeed (as we describe in detail in chapter 4) the results achived using the elastic filter were much better.

#### 2.3.1.2 Removing constraints

## 2.4 Blame system

In order to make boa useful we aim not only to inform the programmer about possible overruns, but also to direct him into the problematic code, which made boa generate the warning in the first place. We call this feature *blame,* since it allows boa to blame specific operatonios in specific source lines, allowing the programmer to examine each warning quickly and understand whether there is a real overrun in the code or the warning is a false alarm.

As far as we know, previous works in this area did not deal with this issue ([4]) or used third party tool in order to provide the user with all of the code references to the suspect buffer ([1]).

The input for our blame system is a list of *unsafe buffers*, buffers in which the linear solver detected a possible overrun. For each of the unsafe buffers the output is a set of constraints, this set should be the minimal nessecary set of constraints for the generation of the overrun.

Note that the definition of this set resembles the definition of an IIS - in both cases we look for a minimal set of constraints which imply a global property of the constraints problem. Therefore we based or blaming system on reducing the blaming problem to the problem of finding an IIS, and then using the existing algorithms to find the desired set of constraints.

## 2.5 Implementation

# Chapter 3

# Development Process

## 3.1 Initial Requirements

We set out to develop Boa by first defining a preliminary set of requirements:

- Provide analysis over any valid[1] C code as is, without requiring the programmer to provide any meta information.
- Provide *Soundness*: Report 100% of the buffer overruns in the code, with no *False Negatives* reported.

## 3.2 Lenient Assumptions

In order to achieve these requirements we also defined lenient assumption on the input code:

- The programmer knows that the C string library requires that a string must end with the NUL terminating character '\0', and will never mutate the last byte of a buffer in a way that will cause an overrun.
- The programmer never uses an uninitialized value.

## 3.3 Research and Technology Survey

After defining the initial requirements we began researching past work on the subject of buffer overrun static analysis, and set out to find which tools already exist that can be used to build Boa.

Our research reached the conclusion of using a linear problem solver, solving a set of linear constraints generated from each instruction in the source code, in order to find buffer overruns, as described in chapter 2.

We set a goal of using only open source tools which are publically available for free use under an Open-Source compatible license, and converged towards the use of the following tools:

- GLPK - the GNU Linear Programming Kit [http://www.gnu.org/software/glpk/glpk.html], available under version 3 of the GNU Public License. Used to solve the linear constraints using the Simplex Algorithm.
- The Clang C Front-End [http://clang.llvm.org/], available under the University of Illinois/NCSA Open Source License. We planned on using Clang's plug-in system in order to go over the code's Abstract Syntax Tree and generate linear constraints according to the instructions in the code.

We then set out to create prototypes. First we created example, "*Hello World*" style, programs that make use of Clang and GLPK's interfaces separately,

[1] We define *Valid C* code as any code that is compiled by *gcc* with the *-Wall* flag without any warnings.

## 3.4 Test system

## 3.5 Version control

### 3.5.1 Code reviews

# Chapter 4

# Results

We tested boa on several widespread real world programs. We tested to see whether boa discovers 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 are availble in boa git repository[2].

Table 4.1 summarizes the performance of boa on several programs, the reported running times are the results of experiments ran on a 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 | ssh |
|---|---|---|---|---|
| Source lines | 230 | | 332 | 3483 |
| Constraints | 2894 | | 1206 | 7642 |
| Running time | | | | 2.608s |
| Running time (blame) | 2.508s | | 1.304s | 20.049s |
| Buffers | 34 | | 15 | 92 |
| Overruns reported | 6 | | 8 | |
| Real overruns | 1 | | 1 | 0 |

Table 4.1: boa performance on various real world examples

Over the next sections we will present and describe in details some of the possible overruns boa detected in these programs, and use these examples to demonstrate how boa can be used to detect overruns and how one can analyze false alarms.

## 4.1 fingerd

We tested boa using *fingerd*, unix finger deamon. 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 1024[1] bytes buffer *line*. As far as we know, this is the only real buffer overrun in the 230 lines of source code.

Running on *fingerd* source, boa reported overruns on 6 out of the 34 buffers. Next we present boa's blame for 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 ...  ]
```

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

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.

**rhost** is a char buffer meant 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 their 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 allocates 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 incerementai pointer aliasing will be limited to the buffer size - and therefore boa reports 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

[4] A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities