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

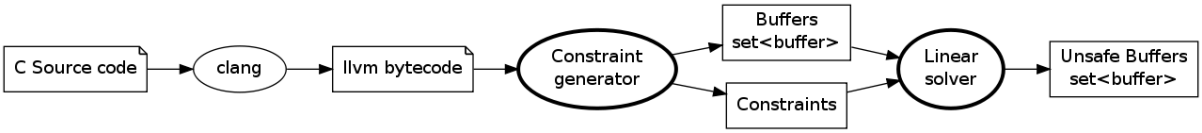

Figure 2.1: Main components and stages

2.2 Constraint Generator

2.2.1 Integers

2.2.2 Direct array access

```
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 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 -

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

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 $0 \ldots n-1$. 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 -

```
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 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[1]. 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 ([3]) or used third party tool in order to provide the user with all of the code references to the suspect buffer ([2]).

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 our 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.

The reduction itself simply involves adding two constraints to the problem -

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

These are the two constraints we use to ensure that all of the buffer accesses are safe, with one different - in this case the left hand expressions are constraint variables, and the right hand are values - 0 in the second constraint and **the value** assigned to {buf!alloc!min} in the first solution of the constraint problem. That way the problem become infeasible, and removing the infesibility will ensure safe only accesses to the buffer left.

During the constraint generation proccess we add each constraint a *blame string*, which contain a brief explanation about the nature of the command caused this constraint (e.g. "malloc call", "add instruction" etc) and it's source location (source file name and line number). Once the elastic filter return the set of constraints which cause the infeasibility (an therefore - cause the overrun) we print the *blame strings* of these constraints.

2.4.1 Constraint priorities

In order to provide the user with the most relevant blame results, we introduce three levels of constrainst -

Structural constraints which model the relations between {buf!read}, {buf!used} and {buf!write}. These constraints are part of boa's internal implementation (disscused in details on section 2.2.4), and does not supply the user with information about his possible buffer overrun.

Buffer aliasing constraints disscussed in details on section 2.2.4, which usually does not cause an overrun by themselves, but might be very important for the user to understand how does writing to a pointer *ptr* cause an overrun on buffer buf.

Normal constrainst are any other constraints, involving integer arithmetics, library function calls etc.

The cause of the overrun is usually a *normal constraint*, while *buffer aliasing* constrains might be vital for the understanding the connection between the line where the overrun actually happens and the buffer (decleration) itself. *Structural constraints*, on the other hand, will always be part of an IIS, but the are irrelevant for the user.

When we run the elstic filter in order to discover the blaming set we first allow the elstic filter to choose only *normal constraints*, and than run the elastic filter again, allowing it to choose *aliasing constrainst* as well. That way we ensure that the most relelvant blame results will appear first, the aliasing blames will also appear, but the *structural constraints* will not distruct the user from the actual problem in his source code. A similar approch is taken when we use the elstic filter to turn the initial infeasible problem into a feasible problem - in that case we allow the elastic filter to remove *normal* or *aliasing* constraints, but not *structural* constrains, since the latter reflect an internal structure we intend to keep - and can not be the cause of the overrun.

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.
- Provide output in a machine readable format that is also easily readable by humans.

We also had a goal of minimizing false positives, for which we couldn't define a measurable requirement at the initial stage.

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[2], available under version 3 of the GNU Public License. Used to solve the linear constraints using the Simplex Algorithm.

[1] We define *Valid C* code as any code that is compiled by *gcc* with the *-Wall* flag without any warnings.
[2] GLPK - http://www.gnu.org/software/glpk/glpk.html

- The Clang C Front-End[3], 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, and then combined the two prototypes to a single system that works according to our system design, produces constraints by traversing the source code, solves the constraints system and reports buffer overruns accordingly.

At this stage we created constraints such that any buffer access and use and any function call created an overrun on all buffers involved. This trivial handling already meets our well defined requirements of handling any C source code and being *Sound*, and from this point we began iterating on reducing false positives.

This methodology allowed us to have at any given point a sound working system which gained quality as the construction progressed.

3.4 Test system

To facilitate the construction of Boa, we worked using *Test Driven Development.*

3.4.1 Black Box Testing

Since our entire system is hosted as a Clang plugin, for most purposes *White Box* testing was not reasonable for our purposes.

We therefore created a *Black Box* testing system which allowed us to write tests for the full system behaviour.

A test case is comprised of a source code (.c file) and assertions (.asserts file), which define which buffers should be marked as having overruns, and which buffers should be marked as safe. Buffers may be defined by their name, by their source code location, or by both.

Example:

safe.c:

```
1 #include <stdio.h>
2
3 int main() {
4   char cString[] = "I am safe!";
5   char overflow[8192];
6   puts(cString);
7   gets(overflow);
8   return 0;
9 }
```

safe.asserts:

```
1 HAS ByName overflow
2 NOT ByName cString
3 BLAME ByName overflow unsafe function call gets
4 BLAME ByLocation tests/testcases/safe.c:7 gets
```

We then wrapped this testing system with scripts that run the tests over all available test cases and indicate failure if any test case failed, and incorporated this script into our makefile, such that running *make tests* compiles binaries as necessary and runs the tests on the result. This allows us to run all our tests after every change in the code, thus giving us confidence that old behaviour is not broken, and allowing us to start developing a feature by first defining a failing test case, then writing code until the test passes.

We later expanded the testing system to verify the existence and content of *Blame* information per a given buffer.

[3]Clang C Front-End - http://clang.llvm.org/

3.4.2 Unit Testing

For those cases where we could generalize common behaviour, we also wrote *White Box* unit tests.

We used *Google C++ Testing Framework* (commonly known as *googletest*) as the framework that runs the tests so we could focus on writing the tests. In some cases we made adjustments to our code so that internal Boa classes could be subclassed.

3.5 Version Control

We used *git* as our versioning control system. We leveraged the distributed nature of git and the cheap branching of git to create a workflow based on *feature branches*, where each of us worked on a feature on a separate branch of the code (usually working on more than one feature at a time), which was later merged into our integration branch, *"master"*.

3.5.1 Online Hosting

We used Github as an online code repository. This also gave us basic project management tools such as issue tracking and code reviews.

3.5.2 Code Reviews

Github allows developers to send each other code reviews (named *Pull Requests* in git terminology). We embraced this feature into our work flow such that all of the code that goes into *master* has to be reviewed by a team member.

For each code review, one of the team members which did not write the code would read the code and a discussion would start, where the reviewer requested improvement to the code, tests or documentation such that the code is correct and readable. By using this workflow, we ensured that the final code is not only correct, but also readable, since each code snippet had to be read by a programmer that did not see the code being written before being integrated into *master*.

3.6 Construction

During the Construction phase we used the following workflow:

- Create a local branch which is a copy of *master*.
- Write a failing test case.
- Write code that handles the failures of the test case, detects all possible overruns and does not report a false positive where not needed, until the test passes.
- Upload the branch to the online repository.
- Send a *Pull Request* to the team.
- Repeat.

Git's distributed nature allowed us to use this workflow with minimal overhead - On the one hand, orthogonal changes could be developed in parallel, with one programmer usually working on more than one change simultaneously in different branches. On the other hand, when programmer A needed to base his changes on code written by programmer B which was not yet reviewed and integrated, he could pull the work-in-progress branch from A's repository and base his own work without being blocked by the review process. *Git* also makes it easy to later merge additional changes made by programmer B (for example, as a result of the code review process).

3.7 Porting To LLVM

After we developed some initial features such as generating constraints according to integer arithmetic instructions, we began seeing some difficulties with our use of the Clang front-end as our constraint generation mechanism:

- The representation of various instructions are scattered across many nodes in Clang's representation of the AST, which made it hard to write generalized code that handles multiple cases.
- Clang's *RecursiveASTVisitor* implementation has only been written recently, and as cutting edge code it proved to be somewhat uncomfortable to work with.

Also, working directly on the AST of a high level language meant that the space of instructions that Boa needs to handle is quite large.

We therefore decided it would be best to first compile the code into LLVM bitcode, and then generate constraints according to that bitcode using an *LLVM Pass*.

We first prototyped an LLVM pass in the same manner we prototyped Clang, and then rebased Boa to use LLVM instead of clang in its architecture. All of our test cases were still valid (and at this stage, failing), so we continued our construction cycle, iterating until all existing test cases pass, usually using the previously written code as a reference as to how to handle a certain instruction (e.g., the constraints generated for a multiplication of two integers).

Within two weeks all our existing test cases passed, and we could refocus on further reducing false positives.

Having specific test cases defining the required handling of specific C instructions, a modular design where the mathematical work surrounding solving a linear constraint problem is separated from the handling of source code instructions, and the encapsulation of our algorithmic code all allowed us to very easily handle this major change in our system architecture, without having to rewrite the entire system from scratch.

3.8 Additional Requirements

As work on Boa progressed and enough bitcode instructions were properly handled, we could begin testing against code snippets from real world code. It soon became clear that it is not enough to notify the programmer which buffers are overrun, and Boa should also supply the programmer with information as to which code-level statements caused the buffer to be overrun.

Also, we noticed that most of the operations on character buffers in the real world are using library functions from *string.h* and *stdio.h*, and not direct array access, and that a common cause for buffer overruns is the use of *unsafe* library functions which do not use information about the length of the buffer, and may cause an overrun.

We therefore defined the following additional requirements:

- Provide machine readable *Blame* information which is also easily readable by humans.
- Model 100% of the functions in *string.h* and *stdio.h* such that all safe accesses will not be marked as unsafe, and all unsafe uses of library functions will mark the involved buffers as having overruns.
- Provide blame information per unsafe library functions such that if an unsafe function is used, the blame information will explicitly contain a warning against that specific function and will provide the source location of the unsafe function call.

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[1].

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	md5
Source lines	230		332	3483	432
Constraints	2894		1206	7642	
Running time				2.608s	
Running time (blame)	2.508s		1.304s	20.049s	
Buffers	34		15	92	2
Overruns reported	6		8		2
Real overruns	1		1	0	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[2] 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] Boa's git repository - https://github.com/tzafrir/boa

[2] 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)
142         logerr("malloc: %s", strerror(errno));
```

...

```
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 inceremental pointer aliasing will be limited to the buffer size - and therefore boa reports a possible buffer overrun.

4.2 flex

4.3 MD5

We have tested boa against *RSA Data Security, Inc.* implementation of *MD5*, the a well known cryptographic hash function. Since MD5 is a cryptographic algorithm, the implementation consist of a lot of bitwise operations on

Bibliography

[1] Chinneck, J. W., and Dravnieks, E. W. Locating minimal infeasible constraint sets in linear programs. *INFORMS Journal on Computing 3*, 2 (1991), 157–168.

[2] Ganapathy, V., Jha, S., Chandler, D., Melski, D., and Vitek, D. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM conference on Computer and communications security* (New York, NY, USA, 2003), CCS '03, ACM, pp. 345–354.

[3] Wagner, D., Foster, J. S., Brewer, E. A., and Aiken, A. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Network and Distributed System Security Symposium* (San Diego, CA, Feb. 2000), pp. 3–17.