Project in advanced programming - Outline

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

November 9, 2010

We intend to implement a static Buffer Overrun Analyzer (boa). Boa will receive a C program as input, and determine whether a buffer overrun is possible during execution of the code. This must be done with care - while most static analysis methods result in some false positives, extra care should be taken to ensure no false negatives (measures not taken in [1, 2]). We refer to the lack of false negatives as the *soundness* of the analysis.

Boa will be implemented in two stages, the first will be context and flow insensitive analysis. However, context and flow insensitivity is prone to false alarms. Therefore, we intend to test limited flow and context sensitivity, which we hope will be able to reduce the amount of false positives without leading to false negative results.

Pointer analysis is a main phase of boa. We will begin with a naive points-to analysis, in which every pointer is suspect of aliasing any other pointer. Once preliminary results are achieved, we will attempt to refine this analysis while keeping it sound. We will explore the algorithm presented in [4] as a possible basis for this.

Assumptions:

- No concurrency (for any flow sensitive analysis)
- No unions

We intend on hooking into Clang's [5] static analysis run and create constraints on integers and buffers while Clang parses the source code. Clang API will be used to generate integer linear programming constraints for each buffer and integer in the code. These constriants will model the max and min used (and allocated) index for each buffer. Finally we will use GLPK[6, 7] to solve the integer linear programming problem designated by the constriants, and the solution will determine whether buffer overrun is possible.

If time allows, we will also explore the idea introduced in [3]. By performing symbolic execution on the sites of the possible buffer overflows, we can further differentiate between false positives and true vulnerabilities - improving the overall quality of the analysis.

Basically, this includes:

1. Extracting the relevant code snippet. This cannot be done perfectly (halting problem anyone?), but this CAN be done conservatively. In their research, Kim, Lee et al. limited this backtracking to within the context of the procedure (assuring a linear number of execution paths). We will investigate, among other things, if there is some way to expand this while retaining scalability, and if this results in a measurable gain in filtering false alarms.

2. Translating the snippet into SMT formulae, expressing the constraints in quantifier-free FOL. This is simple for most statements, but requires special care regarding loop blocks and function calls.

   (a) Loop blocks are unwound, examining every possible (and feasible) execution path until either the loop cannot execute (i.e. the post-condition must be satisfied) or a buffer overrun cannot be

discredited in some iteration. This is done recursively, and as far as we can tell from a cursory examination – this algorithm cannot be assured to always stop. We will investigate this, and possibly add some safeguards to prevent stack overflow.

(b) Functions calls are not executed symbolically. Instead, the constraints generated from the context insensitive analysis are used for scalar return values. We will also examine this.

In order to simplify this process, the source code is translated into a subset of C, in which there is only one type of loop, and statements have no side effects. While it is possible to translate 'goto' statements into this language, this adds an additional complexity (possible need for program flow re-structure).

We plan to perform two stages of testing:

1. Very simple test cases in order to check basic functionality - No buffers at all, safe access to buffers, some unsafe access and C library functions (strcpy, strlen, etc.), dynamically allocated buffers, some aliasing. These will be relatively small programs, written by us, in order to execute quickly (unit testing).

2. Real world open-source programs, such as those used in the referenced articles (wu-ftpd, sendmail, bind etc.). This will enable us to test our project against real-world challenges and also enable comparison with previous projects' benchmarks.

References

[1] Buffer Overrun Detection using Linear Programming and Static Analysis

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

[3] Filtering False Alarms of Buffer Overflow Analysis Using SMT Solvers

[4] CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C

[5] http://clang.llvm.org/

[6] http://en.wikipedia.org/wiki/GNU_Linear_Programming_Kit

[7] http://www.gnu.org/software/glpk/