OVS Assignment: Finding Security Vulnerabilities with CBMC

Jan Tobias Mühlberg and Frank Piessens

18th November 2011

1 Introduction

The purpose of this assignment is to make you familiar with state-of-the-art software verification tools and their use for security verification and testing. In particular, you will employ a bounded model checker, CBMC, in a case study on finding vulnerabilities in a simple address book application that is written in C.

Software model checking is a technique for proving that a model of a software system satisfies a given property by exhaustive search. That is, the model checking algorithm enumerates all states of the model while searching for a state such that the property is violated. There exist a range of modern software model checkers that automate the task of deducing models and properties from a given software.

When being applied to real-world software, model checking quickly suffers from combinatorial state explosion. Bounded model checking is a technique to tackle this problem by restricting the "depth" to which the state space of the program is constructed.

CBMC[1] is a bounded model checker for ANSI-C and C++ programs. It can be used to verify that a program does not contain buffer overflow errors, pointer safety violations and that user-specified assertions hold. CBMC performs verification by unwinding the loops in the program, translating the resulting paths into equations, and passing these equations to a decision procedure.

2 Assignment

The software to be analysed implements the backend of an address book application for embedded Linux devices. It facilities creating, managing and browsing a list of contacts. It does not provide an elaborate user interface, which is to be implemented separately and out of scope for this assignment.

[1] `http://www.cprover.org/cbmc/`

It also does not implement facilities for writing and reading the contacts to and from persistent storage. That is, the address book is intended to run infinitely and its runtime environment ensures that the program's state is stored persistently when needed.

Thus, the address book has to satisfy a number of stringent requirements with respect to safety and security:

1. The program shall not crash.
2. The program shall not leak memory.
3. It shall not be possible to exploit the program, through direct interaction or via a user interface, to modify the program's behaviour or to gain access to other components of the device.

To complete the assignment you have to employ CBMC to find violations of the above properties in the program, demonstrate how these violations may be exploited, fix the program, and discuss your findings.

2.1 Mandatory Tasks

Task 1: Apply CBMC to check the address book application for the absence of `NULL` pointer dereferences. In particular, check that the result of an `malloc` is never dereferenced without a runtime-test ensuring that a non-`NULL` pointer has been returned. To accomplish this task you have to instrument the `malloc` function and then add `assert` statements to the code. Report and fix all errors that you identify. Document and discuss your proceeding and your results.

Task 2: Implement a stack of integers using a singly linked list in C. Your program shall consist of (at least) a function `push`, a function `pop`, and a function `main`. `push` and `pop` shall facilitate adding and removing elements from the stack, respectively. In `main` you shall implement a test harness so as to thoroughly test your stack for a large number (> 1000) of elements. Test your program. Now employ CBMC to check the program for the absence of `NULL` pointer dereferences, memory leaks, buffer overflows and pointer safety errors. Use CBMC's `--bounds-check` and `--pointer-check` options in combination with your instrumentation of `malloc` from Task 1. Apply the two options separately and together; experiment with different loop bounds. Report and discuss your findings.

Task 3: The address book application uses a doubly-linked list to store address data. Extract the list implementation from `addrbook.c` and verify it for the same safety properties as in Task 2. Report and discuss your findings.

Task 4: Apply CBMC to check the address book application for the absence of memory leaks, buffer overflows and pointer safety errors by using CBMC's `--bounds-check` and `--pointer-check` options. Report and fix all errors that you identify. Document and discuss your proceeding and your results.

Task 5: Give coding guidelines that you would impose on your development team to fully exploit the potential of automated verification tools such as CBMC. Discuss why your guidelines are useful.

2.2 Optional Tasks

Task 6: Familiarise yourself with the idea of format string vulnerabilities. We recommend reading "Buffer overflow and format string overflow vulnerabilities" by Lhee and Chapin (appeared in *Software: Practice and Experience*, volume 33, issue 5, pages 423–460, 2003, John Wiley & Sons). Experiment with CBMC to develop a strategy for using the tool to find format string vulnerabilities in a given program. Your approach may involve program instrumentation as well as restrictive coding standards. Demonstrate your idea in terms of one or more example programs.

Task 7: Apply CBMC to check that the address book application does not contain any vulnerabilities through uncontrolled format strings (i.e., for the `printf` family of libc functions). Report and fix all errors that you identify. Document and discuss your proceeding and your results.

Task 8: Demonstrate how to exploit a format string vulnerability in the address book application to *(i)* crash the program and *(ii)* to gain access to the underlying operating system. Your demonstrator may be written in any language you like. Discuss your proceeding and your results.

Task 9: Compare the CBMC approach to other formal verification techniques and to software testing approaches that you are familiar with.

3 Practicalities

The project has to be worked on in groups of two to three students, of whom each is supposed to spend 30 hours on the tasks. If you do not manage to finish all tasks within this time you should instead discuss and reflect on how you would solve the open tasks and what issues you have encountered.

This assignment is handed out on 18th of November 2011. The assignment is due on 23rd of December 2011. Your results shall be submitted to Tobias Mühlberg and Frank Piessens by uploading a project report and related material to *Toledo*. We expect you to upload a `tar` (or `Zip`) archive containing:

1. A short report (PDF file, max. 10 pages written in English) containing your documentation and discussion for each of the tasks.

2. A number of directories named `task_<n>` so that each directory contains the results for task number `<n>`. For tasks 1, 3, 4 and 7 we expect that the folder contains a fixed version of the program that verifies in CBMC. For tasks 2 and 6 your example programs shall be placed in the respective folders. For task 8 the folder shall contain the two exploit demonstrators. You may add `make` or `bash` scripts to each folder so as to facilitate compilation, running CBMC, or executing the demonstrators.

   Document the content of your directories in the report. Give instructions on how to invoke the exploit demonstrators and CBMC on the different files (i.e., loop bounds and other parameters). Also document what the expected result of running CBMC on the different files is: CBMC might fail intentionally on some example programs. It may unintentionally fail on your modified version of the address book but you might have an indication or explanation for us.

Good luck!

4 Material

The following materials may be useful for you to complete the tasks:

- Course notes, this assignment text and the address book application are available on *Toledo*
- There will be a *Q&A Session* on the 2nd of December
- Use the course forums on *Toledo* for further questions
- CBMC website with tutorials and documentation: `http://www.cprover.org/cbmc/`
- CBMC manual: `http://www.cprover.org/cbmc/doc/manual.pdf`
- A set of slides on CBMC: `http://www.cprover.org/cbmc/doc/cbmc-slides.pdf`
- Lhee and Chapin. Buffer overflow and format string overflow vulnerabilities: `http://dx.doi.org/10.1002/spe.515`