Memory Safety Bug Hunting with BAP

Tiffany Bao (tiffanybao@cmu.edu)
Daming Chen (ddchen@cmu.edu)
Rijnard van Tonder (rvt@cmu.edu)

http://ddcc.github.io/15745-s15-project

15-745 S15 Optimizing Compilers
March 17th, 2015

Although numerous debugging tools have been developed for performing static and dynamic software analysis, a large number of these require access to source code or utilization of special compile-time flags to embed debug information within the output binary. In contrast, little effort has been spent on developing static binary analysis tools which can directly analyze off-the-shelf executables without executing them. For this project, we plan to focus on a subset of common programming bugs that tend to introduce security vulnerabilities, known as memory-safety errors. Examples of these include buffer overflows, double frees, and uninitialized variables. Over the course of the project, we will develop and implement various static memory-safety analyses into our group's Binary Analysis Platform (BAP)[1]. BAP is a pre-existing framework written in OCaml for performing analysis of binary code, and we'll be using it primarily with the ARM architecture.

- Getting Started
  - All project members have had previous experience writing analyses with the Binary Analysis Platform (BAP), including a basic context-sensitive interprocedural analysis for generating call strings
- Project Plan
  - 75%
    - Implement basic intraprocedural static instruction register tracking

      Function return values and variables are often passed through various registers or memory locations, making them difficult to track in the disassembly. A possible solution is to implement a mechanism similar to taint analysis for tagging registers and memory locations with the variables that they correspond to, allowing higher-level analyses to be performed.

    - Implement intraprocedural detection of unchecked memory allocations

[1] https://github.com/BinaryAnalysisPlatform/bap

In this step, we start our checking at function malloc(), and assert that the validness of the return value of malloc() should always be checked. For example, suppose we have a binary with source code shown below:

```
#include <stdlib.h>
#include <stdio.h>

int main() {
  // Uninitialized variable
  char *array = 0;


  array = malloc(32 * sizeof(*array));

  if (!array)
    return -1;

  free(array);

  return 0;
}
```

The red part of code shows that after array is allocated by function malloc() one should always check if array is a valid pointer or not. This is a reasonable assertion, especially from security perspective, because if array is invalid attacker could utilize the variable to exploit.

Therefore, we will start looking at function malloc, and to check if a malloc is followed by return value check in the binary.

- 100%
  - Implement basic intraprocedural detection of buffer overflows in stack

    The maximum size of stack variables can be inferred based on changes to the stack pointer at the prologue and epilogue of a function. Then, loads and stores to stack variables within the function can be checked to ensure that they do not exceed the stack allocation.

  - Implement basic intraprocedural detection of buffer overflows in heap

    The size of heap variables can be inferred based on arguments provided to memory allocation functions such as malloc(). Then, loads and stores to heap variables within the function can be checked to ensure that they do not exceed the stack allocation.

    - 125%
        - Extend analyses to be interprocedural
        - Implement intraprocedural detection of null pointer dereferences
        - Implement detection of dynamic memory errors; e.g. dangling pointers, double frees, and invalid frees
        - Implement intraprocedural detection of variables with uninitialized variables; e.g. usage of variables with unassigned values
    - 200%
        - Extend analyses to be path-sensitive
        - Implement basic flow and context-insensitive pointer analysis for identifying aliasing pointers to handle C++ binaries
        - Evaluate soundness, completeness, and performance of our tool against similar bug checking tools, e.g. Address Sanitizer, Valgrind MemCheck, Clang Static Analyzer, Coverity, CodeSonar, etc.
- Schedule

Week	Goal
3/16 - 3/20	Write project proposal
3/23 - 3/27	Work on static instruction register tracking
3/30 - 4/3	Work on static instruction register tracking, work on unchecked memory allocation detection
4/6 - 4/10	Work on unchecked memory allocation detection
4/13 - 4/17	Work on buffer overflow detection, write milestone report
4/20 - 4/24	Work on buffer overflow detection
4/27 - 4/29	Write final report, prepare for poster session

- Milestone
    - By the milestone due date (4/16), we plan to have implemented all of our 75% goals, namely intraprocedural detection of buffer overflows in both the stack and heap, and interprocedural detection of uninitialized variables and unchecked memory allocations.
- Resources
    - Since the Binary Analysis Platform (BAP) is open source, all project members (and members of the public) have access to the software. No additional hardware is necessary for this project. The majority of the pre-existing tools that

we plan to compare our analysis with is open-source, however, certain closed-source tools may require negotiations for access.

- Literature
    - David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. **BAP: a binary analysis platform**. In *Proceedings of the 23rd international conference on Computer aided verification* (CAV'11), Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer-Verlag, Berlin, Heidelberg, 463-469.
    - Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. **Frama-C: a software analysis perspective**. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods* (SEFM'12), George Eleftherakis, Mike Hinchey, and Mike Holcombe (Eds.). Springer-Verlag, Berlin, Heidelberg, 233-247.
    - Susan Horwitz, Thomas Reps, and David Binkley. 1990. **Interprocedural slicing using dependence graphs**. *ACM Trans. Program. Lang. Syst.* 12, 1 (January 1990), 26-60.
    - Johannes Kinder, Helmut Veith. **Jakstab: A Static Analysis Platform for Binaries**. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008)*, vol. 5123, Lecture Notes in Computer Science, Springer, July 2008, pp. 423–427.
    - Bjarne Steensgaard. 1996. **Points-to analysis in almost linear time**. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (POPL '96). ACM, New York, NY, USA, 32-41.