Lehrstuhl für Sicherheit in der Informatik
Prof. Dr. Claudia Eckert

LLVM-based Detection of Integer Overflows

M.Sc. thesis problem statement

Ever wanted to use reliable software in which spy software can not be implanted? Well, you are just right, if you want to develop a LLVM-based tool which can mitigate a major source of memory corruptions [4] (i.e., integer overflows (IOs)).

IO detection is an unsolved problem since more than 30 years. Real live software (e.g., Linux kernel, Chrome browser, etc.) suffers heavily from these types of memory corruptions, mostly because of the unsafe used programming languages (e.g., C/C++). Furthermore, some of the IOs can be exploited in order to perform advanced code reuse attacks or *implant* malicious software into software systems.

The goal of this thesis is to develop an integer overflow detection tool which can detect IO bugs in C/C++ statically through code compilation and which has better accuracy than existing stat-of-the-art tools, IOC [2], IntFlow [3] and IntEQ [1]. The tool should be based on static information flow tracking and dynamic program analysis and reuse the information flow tracking used in the previous tools.

First, we want to extend the scope of the static analysis (i.e., make the analysis interprocedural) during LLVM link time through a LLVM pass which can extract paths which span multiple files and compare them in order to filter out harmful from benign paths (i.e., contain an integer overflow bug or not). Second, loops should be treated with care—for example—loops can be unrolled with varying number of depths (e.g., 1-1k-10k-100k-1000k) in order to increase precision of our tool. Third, chains of sinks and sources (e.g., sink—source—sink—...) can be analyzed in order to determine complex relations between those program locations. Finally, other techniques can be imagined and implemented in order to improve the precision of the developed tool.

The tool will be evaluated using the Linux kernel, web browsers and the SPEC CPU 2006 benchmark. Finally, we will compare our tool w.r.t. the previously introduced state-of-the-art tools.

Requirements

Good C/C++ programming skills, LLVM pass knowledge is beneficial

Contact

Paul Muntean, M.Sc. E-Mail: paul@sec.in.tum.de, Tel.: (089) 289-18566, Tender date: January 17, 2017, Beginning: now

Work Plan

1. Develop knowledge of state-of-the-art IO detection tools:
   (a) Read provided references and find related work on this topic.
   (b) Test the IOC, IntEQ and IntFlow tools.
   (c) Write a state-of-the-art survey (max 5, A4 pages), which presents and compares the investigated techniques and tools.
2. Perform a security analysis of the software evaluated in the references:
   (a) Identify alternative binary or software protection techniques against IO bugs.
   (b) Identify software which contains IO bugs which can be exploited to perform code reuse attacks.
   (c) Asses the attack surface reduction in software which was analyzed by the state-of-the-art tools, see references.
3. Implement the IO detection technique(s) which you identified and you think would make sense.
   (a) Choose technique(s) described in literature and/or propose a new technique; argument your choice (e.g. security versus cost trade-off) in written form.
   (b) Implement the chosen technique(s) based on the LLVM framework [1] and document design decisions.
   (c) Note that we provide a LLVM pass on which the tool can be build upon.
4. Evaluation of own implementation and possibly existing tools (case-study):
   (a) Measure effectiveness of exisitng tool against the same programs used in step 2.
   (b) Measure performance, effectiveness, true positives and false negatives of the analyzed software w.r.t.: the SPEC 2006 benchmark, a series of server applications (e.g., Nginx, vfptd, lighttpd, etc.) and web browsers (e.g., Chrome, Firefox, IE) and Linux kernel by recompiling these software.
   (c) Compare the results w.r.t. false negatives, true positives rates of the state-of-the-art tools.
   (d) Analyze and discuss security versus performance trade-offs.
5. The final thesis document must contain:
   (a) Description of the problem and motivation for the chosen approach.
   (b) State-of-the-art survey, including analysis of security and performance.
   (c) Security analysis of the previous mentioned server applications and web browsers.
   (d) Design, rationale for choosing certain technique(s) for implementation.
   (e) Implementation description.
   (f) Evaluation w.r.t. performance, true positives, false negatives, etc.
   (g) Discussion on potential security and performance trade-offs
   (h) Conclusions and future work.

Deliverables

1. Source code of the implementation (i.e., can be implemented as one or multiple LLVM passes) as well as instructions on how to run the tool.
2. Technical report with comprehensive documentation of the implementation, i.e., design decision, architecture description, API description and usage instructions.
3. Final thesis report written in conformance with TUM guidelines.

References

[1] H. Sun, et al. IntEQ: Recognizing Benign Integer Overflows via Equivalence Checking Across Multiple Precisions, In: ICSE 2016. source code: `https://github.com/shqking/inteq`

[2] W. Dietz, et al. Understanding Integer Overflow in C/C++, In: ICSE 2012. source code: `https://github.com/dtzWill/ioc-clang`

[3] M. Pomonis, et al. IntFlow: Improving the Accuracy of Arithmetic Error Detection Using Information Flow Tracking, In: ACSAC 2014. source code: `https://github.com/nettrino/IntFlow`

[4] H. Meer, et al. Memory Corruption Attacks The (almost) Complete History, In: BlackHat 2010. source: `https://media.blackhat.com/bh-us-10/whitepapers/Meer/BlackHat-USA-2010-Meer-History-of-Memory-Corruption-Attacks-wp.pdf`

[1] http://llvm.org/