Jens Knoop, Uwe Zdun (Hrsg.): Software Engineering 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 43

Learning how to prevent return-oriented programming efficiently

David Pfaff [1] Sebastian Hack[1] Christian Hammer[1]

1 Extended Abstract

The discovery of recent zero-day exploits against Microsoft Word, Adobe Flash Player and Internet Explorer demonstrate that return-oriented programming (ROP) is the most severe threat to software system security. Microsoft's 2013 Software Vulnerability Exploitation trend report found that 73% of all vulnerabilities are exploited via ROP. The core idea of ROP is to exploit the presence of so-called *gadgets*, small instruction sequences ending in a return instruction. By chaining gadgets together, an attacker is able to build complex exploits. The apparent popularity of ROP is explained by its power to bypass most contemporary exploit mitigation mechanisms, such as data execution prevention (DEP) and address space layout randomization (ASLR). DEP and similar page-protection schemes prevent the execution of injected binary code, but ROP re-uses code already present in the executable memory segments, eliminating the need to inject code. ASLR randomizes the location of most libraries and executables, however, finding code segments left in a few statically known locations is often enough to leverage a ROP attack. Since the inception of ROP by Shacham [Sh07], research on ROP resembles an arms race: emerging defense techniques are continuously circumvented by increasingly subtle attacks [CW14].

In our paper [PHH15], we take a novel, statistical approach on detecting ROP programs. Modern microprocessors spend most of their circuits on machinery that optimizes the execution of programs generated by compilers from "high-level" languages. Among this machinery are caches, translation look-aside buffers, branch predictors, and so on. To assist programmers in detecting performance problems, a modern CPU can record several hundred different kinds of micro-architectural events that occur during program execution (e.g. mispredicted branches, L1 cache misses, etc.). These events are counted by the CPU in special registers, the so-called *hardware performance counters* (HPCs).

In this paper, we claim *and experimentally verify* that the execution of a ROP program triggers such hardware events in a significantly different way than a conventional program that has been generated by a compiler. Essentially, micro-architectural events are a side channel by which a ROP program becomes distinguishable from a normal program at run time. There are several considerations that support this hypothesis: First, ROP programs use only indirect jumps (returns) to control the program flow. Common processor heuristics to detect the target of the return are useless in a ROP program because they do not

[1] CISPA, Saarland University, lastname@cs.uni-saarland.de

follow the call/return policy. Second, ROP gadgets are small and scattered all over the code segment. Thus, there is no spatial locality in the executed code which should be observable in counters relevant to the memory subsystem.

We exploit the deviant micro-architectural behavior of ROP programs by training (using existing ROP exploits and benign programs) a support vector machine (SVM) based on profiles of hardware performance counters. Note, that despite our intuition we did *not* short-list any HPC types for training. We receive a classifier to distinguish ROP from benign programs and use it in a *monitor* kernel module that tracks the evolution of the performance counters and classifies them periodically. If the classifier detects a ROP program, defensive actions, like killing the process, can be taken.

We quantitatively evaluate the performance impact of HadROP on benign program runs using the SPEC2006 benchmark: HadROP incurs a run time overhead of 5% on average and of 8% in the worst case. We also establish the effectiveness and practical applicability of HadROP in several case studies that show that HadROP detects and prevents the execution of a ROP payload of an in-the-wild exploit on Adobe Flash Player, 25 new ROP payloads generated by the ROP-payload generator Q that exploit manually injected vulnerabilities in GNU coreutils, Blind ROP [Bi14] of an nginx web server and multiple recent enhancements [CW14, Da14] that allow ROP to bypass previous hardware-assisted detection schemes. HadROP detects and prevents those attacks in any practical scenario.

References

[Bi14] Bittau, Andrea; Belay, Adam; Mashtizadeh, Ali; Mazieres, David; Boneh, Dan: Hacking blind. In: Proceedings of the 35th IEEE Symposium on Security and Privacy, S&P. 2014.

[CW14] Carlini, Nicholas; Wagner, David: ROP is Still Dangerous: Breaking Modern Defenses. In: 23rd USENIX Security Symposium (USENIX Security 14). USENIX Association, San Diego, CA, pp. 385–399, August 2014.

[Da14] Davi, Lucas; Sadeghi, Ahmad-Reza; Lehmann, Daniel; Monrose, Fabian: Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In: 23rd USENIX Security Symposium (USENIX Security 14). USENIX Association, San Diego, CA, pp. 401–416, August 2014.

[PHH15] Pfaff, David; Hack, Sebastian; Hammer, Christian: Learning How to Prevent Return-Oriented Programming Efficiently. In: Engineering Secure Software and Systems, pp. 68–85. Springer, 2015.

[Sh07] Shacham, Hovav: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM conference on Computer and Communications Security. ACM, pp. 552–561, 2007.