Reading memory, without reading memory

Christian W. Otterstad[1]

Department of Informatics, University of Bergen,
Bergen, Norway, April 25, 2016

Extended abstract.

This study is regarding the feasibility of the notion of reading memory indirectly, for the specific purpose of defeating XnR, Execute-no-Read. This research is as of this writing date unfinished.

An attacker facing modern systems with a fine granularity ASLR (Address Space Layout Randomization) implementation requires in many cases a memory disclosure vulnerability or some other memory read primitive in addition to the original exploitable vulnerability.

In classical exploitation the attacker was able to execute arbitrary machine code on the stack and heap, which often were attacker controlled by the very nature of the bug being exploited in the first place. However, with modern mitigation techniques this is no longer feasible for the attacker. In particular, the defender typically employs at least the NX (No-eXecute) bit, ASLR and RAP (Return Address Protection). In special combinations, especially with low granularity ASLR on 32-bit machines, brute force might be sufficient, or if the NX bit is not used for a particular vulnerable memory region, the attacker might still succeed with classical methods.

Exploitation techniques have evolved to heavily employ code reuse. In particular, return-to-libc by Solar Designer was probably the pivotal point towards the modern technique of return oriented programming (ROP). Return oriented programming employs a concatenation of pointers on the stack, allowing the stack pointer to become an instruction pointer. This technique has been shown to be Turing complete for a sufficient set of gadgets, where gadgets are short code sequences terminated by a return instruction. [1] A variant of this is JOP (Jump-Oriented Programming). [2]

However, higher level granularity ASLR prevents the attacker from *a priori* knowing the memory locations of said gadgets. For this reason, the attacker needs a memory reading primitive or information leak to be able to infer these memory locations which are required for the attack.

The ability to read memory prior to launching an exploit egg has therefore become a crucial primitive for modern low-level exploitation. [3][4] A natural mitigation for this issue is to increase the granularity at which memory operations are constrained for the attacker. Akin to hardware protection for disabling execution at page level granularity, the notion of disabling read permission at page level granuarlity has been presented. [5] Execute-no-Read suggests a scheme by which the kernel protects selected memory regions from being read. Effectively preventing any memory read primitive from being iterated or used even once by the attacker. This study attempts to see if XnR can be defeated by indirectly reading memory.

Work in this area has been performed with success before. In particular, the "Braille" project in "Hacking Blind" presents some techniques for doing this. This study attempt to extend these results and use the same techniques to defeat XnR.

Braille works by building only enough gadgets to perform a read and dump the memory to the existing socket. This enables the program to perform *in-situ* memory inspection, irrespective of any diversity, memory permutation or ASLR entropy. However, the specific step of reading memory is prevented by XnR, as pointed out in the original paper. Specifically, XnR recognizes Braille as a threat, but describes it as unlikely to succeed due to the large number of requests to find even a single gadget. It must also be pointed out that the attack models assumes a forking server.

The challenge for the attacker then becomes to find all of the required gadgets without being able to read any memory directly, as performed by Braille.

For this study, an attack program was implemented in C based on the technique described in [4]. The buffer size is simply found by a binary search. Then, similarly to in [6][4] the stack canary, RBP and RIP are found using brute force. Brute force is effective here due to the fact that each byte can be brute forced *individually.* The program then proceeds to find gadgets by brief ROP sequences that are able to determine the presence of stack popping gadgets and eventually perform a syscall through the PLT. The goal is then to perform either a mprotect() to disable the XnR or to perform the full attack by only indirect reads.

The problems and results encountered are particularly that the defender suffers extreme slowdown. This is caused in particular by arc injections that enter non-blocking loops. Each such loop causes a 100% load per core and quickly consumes system resources.

The preliminary results are as of now that the performance issues appear to be an open problem. However, the limitations and overall generalization of the technique is still noteworthy to investigate further.

References

1. H. Shacham, “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*, CCS ’07, (New York, NY, USA), pp. 552–561, ACM, 2007.
2. T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, “Jump-oriented programming: a new class of code-reuse attack,” in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS ’11, (New York, NY, USA), pp. 30–40, ACM, 2011.
3. K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, “Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization,” in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP ’13, (Washington, DC, USA), pp. 574–588, IEEE Computer Society, 2013.
4. A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, “Hacking blind,” in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP ’14, (Washington, DC, USA), pp. 227–242, IEEE Computer Society, 2014.
5. M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, “You can run but you can’t read: Preventing disclosure exploits in executable code,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’14, (New York, NY, USA), pp. 1342–1353, ACM, 2014.
6. H. Marco-Gisbert and I. Ripoll, “On the effectiveness of Full-ASLR on 64-bit Linux.” `http://cybersecurity.upv.es/attacks/offset2lib/offset2lib-presentation.pdf`, 2014. [Online; accessed 18-April-2016].