In Memory Safety, The Soundness Of Attacks Is What Matters*

"Either mathematics is too big for the human mind
or the human mind is more than a machine." –Kurt Gödel

Julien Vanegue

September 4, 2020

Position Statement Large scale software bug detection employs fuzz testing, static analysis and other techniques to identify thousands of new software bugs every year [2] [3] [4] [5]. An unfortunate trend across many large organizations is that the number of bugs found grows faster than the number of bugs fixed. Failure to determine whether bug conditions are sufficient to represent a security vulnerability can result in incorrect prioritization of work. Therefore, software vendors must grow their investment in attack automation to determine real risk and decide which bugs to fix first.

Not All Bugs Are Attacks You are at the bottom of two seemingly endless stairs. [1] On one side lie all *falsely exploitable* bugs, such as buffer overflows with narrow overwrite and restricted byte value range, bugs with data misalignment and overly small injection payload, for which cache desynchronisation, address unpredictability, or low attack bandwidth fail to constitute any realistic leverage for attack. On the other side, the way leads to all *falsely unexploitable* bugs: Read-only out-of-bound accesses big enough to leak secret material, stealth data-only attacks able to overwrite credentials, or bug-combining exploits requiring a yet unknown composition technique... Which direction do you take?

The Bug / Exploit Asymmetry Developing an exploit can be time-consuming, while fixing a bug may only take a few minutes. Exploits are harder to write due to mitigations such as DEP [7], ASLR [8], and heap hardening [9] [10] . Yet, releasing a fix involves development, regression testing, and a long staged roll-out, while running an exploit just takes few seconds. A good balance involves comparing the cost of rolling out new software at large with the amortized cost of attacking targets at scale.

Approximate Exploitability Existing tools [11] [12] over-approximate exploitability *to be on the safe side*. Augmented crash analysis uses mappings such as: Program Counter Control → Must Be Exploitable, Write Primitive → Probably Exploitable, Read Primitive → Probably Not Exploitable, NULL pointer → Not Exploitable. However, bug effects sometimes manifest far away from their root cause. Precise exploitability needs to account for all possible bug effects from the root cause rather than looking at specific executions or crash instances.

Principled Exploitability Approximate Exploitability is neither an optimal strategy for offense nor defense and leads to poor understanding of risk. Benign bugs get fixed in an ill-prioritized way while more impactful bugs may sit in the backlog. [13] Poor bug selection by exploit writers can waste weeks of research on unexploitable bugs. A discipline of exploitability starts with evaluating the hardness of existing techniques in the framework of Exploitability Classes (Figure 1).

Exploit Tools Automated Exploit Generation frameworks [14] [15] [16] have tackled exploit synthesis for stack-based and heap-based buffer overflow exploits. Such techniques do not claim to find an exploit when one is feasible, as they rely on path search and random walks heuristics requiring global reasoning and state space exploration.

Weird Machines A mechanized representation of trust [17] [18] [19] captures exploit execution as synchronized with the target program execution states and transitions. Composition of weird machines illustrates bug chaining where combining distinct bug primitives is needed to craft a complete attack.

Incorrectness Logic A new axiomatic logic [20] [21] promotes under-approximate analysis and local reasoning as a principled approach to bug finding. An error relation is used to track invalidity constraints over program execution. IL can reason about complex software without providing a concrete input or full execution trace of program bugs.

Attack Soundness We introduce a new concept of soundness which is only concerned with the exploitability of a bug known a priori. It is different from the familiar one in program analysis which is a statement about all possible bugs. Exploitability requires a *must-analysis* as the existence of an exploitable program path. It does not require that all program paths are exploitable. Unexploitability requires a *may-analysis* as all program paths may be visited to determine that no bug effect can lead to a successful attack.

The Future Software vendors should determine exploitability of bugs before attackers do. Soundness of attack ensures that a confirmed exploitable bug is fixed before bugs whose exploitability is not proven. Reasoning with exploitability classes will help classify what constitute computable attacks, leading to more efficient bug fixing.

*A wink and a tribute to Patrice Godefroid's original statement [1], also to take with a grain of salt.

[1] Imagine yourself in the computer security edition of the legendary game *Zork* [6]

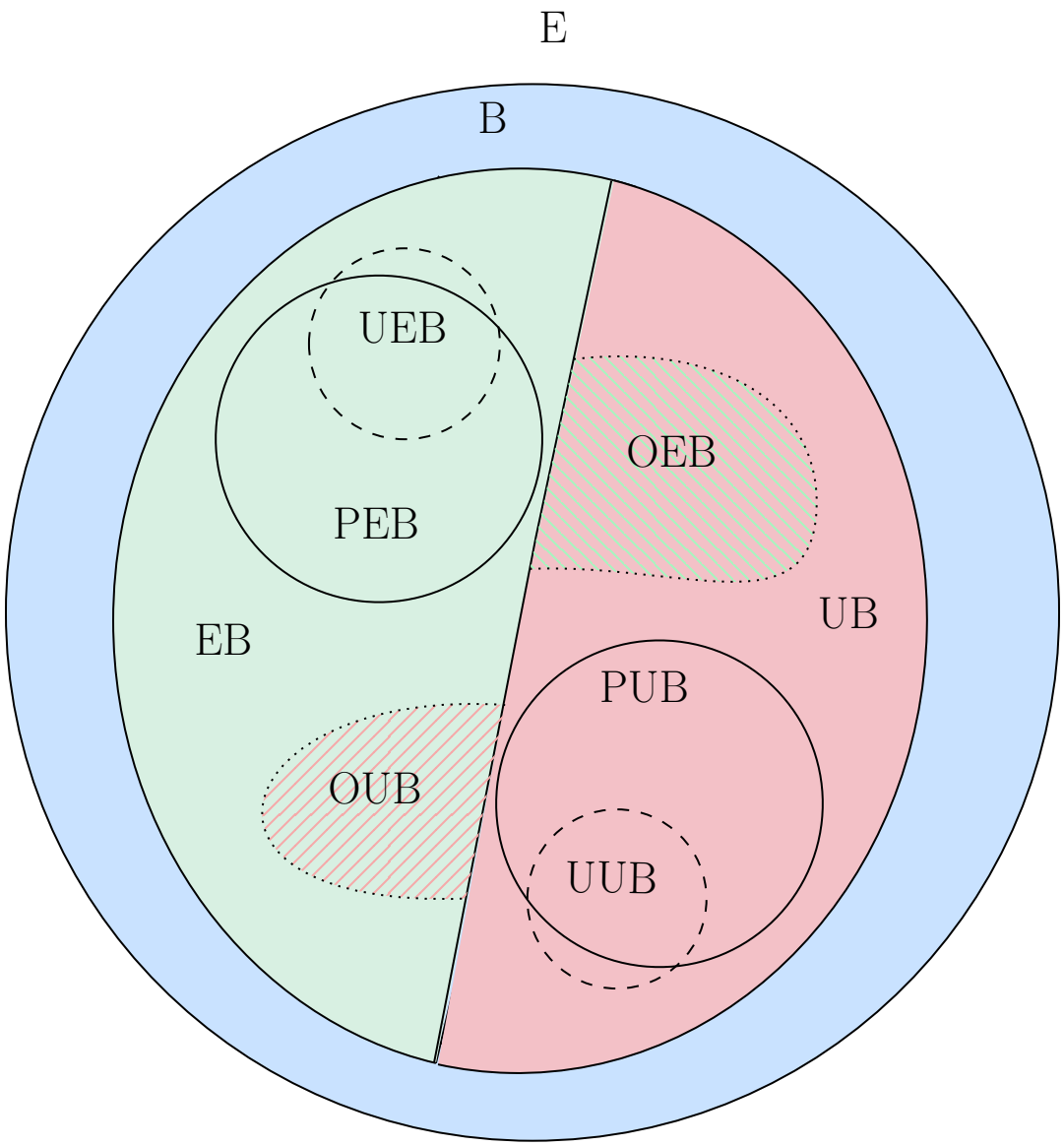

Figure 1: Computational complexity classes of exploitability: B (Bugs), EB (Exploitable Bugs), PEB (Polynomially Exploitable Bugs), UEB (Under-approximate EB), OEB (Over-approximate EB), UB (Unexploitable Bugs), PUB (Polynomially-provable Unexploitable Bugs) UUB (Under-approximate UB), OUB (Over-approximate UB) within E the set of all executions. B = EB + UB, UEB $\subseteq$ EB $\subseteq$ OEB, UUB $\subseteq$ UB $\subseteq$ OUB

References

[1] P. Godefroid, "The soundness of bugs is what matters (position statement)," in *BUGS'2005 (PLDI'2005 Workshop on the Evaluation of Software Defect Detection Tools)*, 2005.

[2] D. Vyukov, "Syzkaller: an unsupervised, coverage-guided kernel fuzzer," 2019.

[3] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing," *Queue*, vol. 10, pp. 20:20–20:27, Jan. 2012.

[4] C. Calcagno and D. Distefano, "Infer: An automatic program verifier for memory safety of c programs," in *NASA Formal Methods* (M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, eds.), (Berlin, Heidelberg), pp. 459–465, Springer Berlin Heidelberg, 2011.

[5] K. Serebryany, "Continuous fuzzing with libfuzzer and addresssanitizer," in *2016 IEEE Cybersecurity Development (SecDev)*, pp. 157–157, 2016.

[6] P. D. Lebling, M. S. Blank, and T. A. Anderson, "Zork: A computerized fantasy simulation game," *IEEE Computer*, vol. 12, no. 4, pp. 51–59, 1979.

[7] S. Andersen and V. Abella, "Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies," 2004.

[8] P. Team, "Pax address space layout randomization (aslr)," 2003.

[9] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda, "Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 88–106, Springer, 2009.

[10] E. D. Berger and B. G. Zorn, "Diehard: probabilistic memory safety for unsafe languages," *Acm sigplan notices*, vol. 41, no. 6, pp. 158–168, 2006.

[11] A. Rank, "!exploitable crash analyzer version 1.6," 2013.

[12] B.-J. Wever, "Bugid - automated bug analysis," 2017.

[13] D. Vyukov, "syzbot and the tale of thousand kernel bugs," *Linux Security Summit*, 2018.

[14] D. Brumley, S. K. Cha, and T. Avgerinos, "Automated exploit generation," Sept. 15 2015. US Patent 9,135,405.

[15] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*, pp. 380–394, IEEE, 2012.

[16] S. Heelan, T. Melham, and D. Kroening, "Automatic heap layout manipulation for exploitation," p. 763–779, 2018.

[17] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, "Exploit programming: From buffer overflows to "weird machines" and theory of computation.," *login Usenix Mag.*, vol. 36, no. 6, 2011.

[18] J. Vanegue, "The weird machines in proof-carrying code," in *2014 IEEE Security and Privacy Workshops*, pp. 209–213, 2014.

[19] T. Dullien, "Weird machines, exploitability, and provable unexploitability," *IEEE Transactions on Emerging Topics in Computing*, vol. 8, no. 2, pp. 391–403, 2020.

[20] P. W. O'Hearn, "Incorrectness logic," *Proc. ACM Program. Lang.*, vol. 4, Dec. 2019.

[21] A. Raad, J. Berdine, H.-H. Dang, D. Dreyer, P. O'Hearn, and J. Villard, "Local reasoning about the presence of bugs: Incorrectness separation logic," in *Computer Aided Verification* (S. K. Lahiri and C. Wang, eds.), (Cham), pp. 225–252, Springer International Publishing, 2020.