[11pt,twocolumn]article

setspace

[normalem]ulem

fancyhdr fancyplain Rais Mense Scientific writing in English — Final version

document Methods Overview of the experiment We compared the results of three commonly used implementations of hardening programs against runtime bugs Serebryany12,Berger06,Bruening11 in order to quantify their effectiveness in dealing with uninitialized reads. Sample Several industry standard tests Berger06 were run in order to reduce the likelihood of any particular test favouring any of the solutions. Each test was run a thousand times in random ordering to eliminate chance advantages or timing specific interference as much as possible. Although the test suite was designed to as heterogeneous as possible it cannot be ruled out with absolute certainty that some test may favour a specific implementation of the solutions under investigation. Sample restrictions The test were only run on a Linux platform. We chose to exclude other platforms from the experiment because Linux is the only platform that all three solutions have native support for. Though alternative support is available making it possible to run the test on other platforms we cannot be certain that these alternative implementations will perform on par with the native implementations of their counterparts. Sampling technique In order to run the test jobs in a randomized fashion we developed our own job scheduler.Doe14 This scheduler allowed us to both randomize the ordering of the test being performed as well as enabling us to record the output. The scheduler recorded the output in JSON format for further processing and analysis at a later stage. Materials The tests were run on Amazon EC2 cloud computing services. M3.medium instances were used, which at the time of writing consisted of a single virtual Xeon E5-2670 CPU, 3.75 GiB of RAM and a 4 GB SSD. Running the test on a platform with different specifications should only affect the real world runtime of the tests but not the relative differences between the results. Procedures The randomized scheduler took care of much of the running of the tests and collection of their results which may fall into one of three categories; correct results where the proposed solution produced the expected output, incorrect results, where a solution is produced but it is an incorrect solution and incomplete results where the program crashes or fails to otherwise produce any results. In addition to the tests exit conditions we recorded the CPU time required to run each test to completion. The results were collected in JSON format and stored to disk on the EC2 instances to be retrieved at the end of the experiment. These results were parsed and imported into R for further analysis. Statistical analysis Statistical analysis was performed in three key areas. First the relative difference between the number of failures were compared for each of the three proposed solutions. The second area for our analysis was the rate of undetected failures. This was established by dividing the number of correct results by the number of incorrect results for each solution. The final area was the runtime for the correct results. This allowed us to compare the overhead introduced by each solution. For the timing measurement the absolute CPU time spent was used in order to minimize fluctuations introduced by environmental factors. Each of the results were baselined against our test results gathered from running the test suite directly on the EC2 instance without any mitigation or detection solutions present. thebibliography10

Serebryany12 Konstantin Serebryany, AddressSanitizer: A Fast Address Sanity Checker. Derek Bruening, Alexander Potapenko, Dmitry Vyukov, Google, 2012.

Berger06 Emery D. Berger, DieHard: Probabilistic Memory Safety for Unsafe Languages, Benjamin G. Zorn, University of Massachusetts Amherst, 2006.

Bruening11 Derek Bruening, Practical Memory Checking with Dr. Memory, Qin Zhao, Google, 2011.

Doe14 John Doe, A randomized task scheduler for fair testing, Does not actually exist, Mars University, 2014.