Performance and Entropy of Various ASLR Implementations

John Detter, Riccardo Mutschlechner

{detter, riccardo}@cs.wisc.edu

December 14, 2015

Abstract

Whether or not a security feature is useful is highly dependent on how effective it is and how it affects system performance. If a security feature is effective but greatly degrades the performance of the system, then the feature is not useful. Likewise, if a security feature is very fast but is not very effective, then it is also not useful. A useful security feature needs to add a reasonable amount of security to the system but at the same time not greatly impose on system performance. In our study, we measure the performance and entropy of ASLR implementations. The implementations we chose are in Debian, HardenedBSD, and FreeBSD with a patch from the HardenedBSD developers. For the most part, our results are not surprising. ASLR has a very marginal impact on performance, while providing excellent security benefits. The distributions in some cases passes a Chi-Squared (χ^2) test, but in some cases also does not. We describe our findings below in more detail.

1 Address Space Layout Randomization

Address space layout randomization (ASLR) is an exploit mitigation technique implemented natively on many modern operating systems including GNU Linux, Mac OS X and Windows. ASLR takes the three parts of the program, the code, stack and heap, and places them at random addresses in the program's address space. The challenge ASLR presents to attackers is that they must now attempt to guess addresses that before would have been known. For example, a common technique for exploiting vulnerabilities is a return-to-libc attack. In order for an attacker to be able to implement a return-to-libc attack, they must know the locations of certain functions in memory such as $system$ or $execv$ [A1]. Since the locations of $system$ and $execv$ are not known to the attacker when the process starts, the

attacker must guess them. When an attacker correctly guesses the address of their desired function and is able to exploit the running process, she is able to do anything at the current privilege level of the running process. However, when the attacker guesses the wrong address, it most commonly causes the process to crash. The reason why ASLR is useful is that it slows down the attacker and forces her to create noise on the local machine or over a network. An Intrusion Detection System (IDS) can be trained to pick up on this noise and notify a system administrator. We have included a sample buffer overflow attack and target for the attack[B3] for readers who are interested an a working example.

ASLR requires generating random bits for changing the position of the program parts. This randomization calculation takes time to compute. This overhead should be considered when discussing the utility of ASLR.

2 Benchmarks Overview

In our research, we benchmarked ASLR implementations of three different common operating systems: FreeBSD, HardenedBSD and Debian Linux. FreeBSD does not support ASLR by default, thus we used a patch that the main HardenedBSD developers Shawn Webb and Oliver Pinter provided[B2]. We used the same hardware for all tests: Intel quad-core i5 3.2GHz processor with 8GB RAM. For the performance tests, we restricted the CPU to a single core to make sure our results were as accurate as possible. We also made sure to fix the CPU frequency so that all tests were run at exactly 3.2GHz. Each operating system was given 50GB of disk space. We did not consider disk storage a major source of error in the tests so we thought it was reasonable to use two partitioned disks for our 3 operating systems. We wanted to include Windows as well, however time did not permit this. We also do not know the APIs as well as we do for Linux/BSD, and thus decided to exclude it for the purpose of this study.

We developed two sets of benchmarks: the first set of tests measures the performance difference (in %, where the actual values are measured in nanoseconds) with ASLR on versus off, and the second set measures the entropy (by showing scatter plots, as well as bits of entropy) in each implementation of ASLR. The performance tests are designed to run operations that should be affected by ASLR computation. The entropy tests are very simple. We just designed one test for each part of the address space randomized: the stack, heap and code.

3 Performance Benchmarks

Our performance tests were chosen based on the randomization operation we wanted to benchmark. The decision to use these tests was based on how and when the address space is randomized. We focused on comparing the difference relative to each system with ASLR on and off, instead of comparing the operating systems against each other. We describe our tests briefly in Figure 1.

Test Number	Program area randomized	function used
1	Code and Stack	$execv$
2	Thread Stack	$pthread\_create$
3	Heap	$mmap$
4	Heap	$malloc$

Figure 1: Performance test descriptions

3.1 Test 1

For our first test we wanted to measure the time it takes to randomize the code region of a running program. We could not find a simple way to measure code randomization without also creating a new stack; therefore, we chose that for our first test we would measure a simple $execv$ call. During the $execv$, the entire address space of the currently running program is replaced by the new executable and a new stack is allocated. When ASLR is enabled, the starting position of both the code and stack are randomized. Usually when a program starts up, the function $\_start$ is called as the entry point to the program. This is part of the C library and its purpose is to set up the environment for the new program. This usually includes allocating data structures on the heap. To avoid this, we built our test program without the standard library. We did have some issues getting the performance test to work on HardenedBSD without the standard library. We weren't able to get a $write$ system call to work without getting a Bus Error (SIBBUS). Therefore, we compiled this test for HardenedBSD with the standard library. We believe that the HardenedBSD percent difference will be slightly higher than reported. It should also be noted that these tests were each ran for one million samples. It took about an hour to run a set of tests for Debian, however it took about 3 hours to run a set of tests on FreeBSD and HardenedBSD. The Linux tests were significantly faster compared to FreeBSD and HardenedBSD; however we are not considering the base speeds of the systems in these tests. We are only trying to compare the performance of the systems with ASLR on and off.

3.1.1 Test 1 results

Linux showed almost no difference between ASLR on and off in this test. HardenedBSD had a small decrease in performance. Because we were unable to run this test without the standard library, the percent difference here is actually probably slightly higher than our result. FreeBSD had a more significant decrease of 4.20%. Because FreeBSD was tested without the standard C library, the difference was from extra time spent in the kernel. We show our results in Figure 2.

	ASLR Disabled	ASLR Enabled	Percent Change
Debian	25766ns	25780ns	0.01%
FreeBSD	79688ns	83031ns	4.20%
HardenedBSD	426399ns	435460ns	2.13%

Figure 2: Performance test 1 results

3.2 Test 2

Our second test measures the time it takes to complete a single $thread\_create$ call. During this call, a new stack with a random start address is added to the current address space. This stack is then used as the stack for the new thread. The thread uses the same address space as the process that created it. Therefore, the code and heap are not randomized during this call. The time reported here is the amount of time that passed in between right before the call to $pthread_create$ and when the thread executes its first couple of instructions. The instruction that come before the final $rdtscp$ instruction include two mov instructions, one $push$ and one sub instruction. These four instructions should be negligible in our calculations.

3.2.1 Test 2 results

There wasn't a significant overhead when running this test on Linux or HardenedBSD. However the difference on FreeBSD was very significant. We observed a difference of 11.72% when testing FreeBSD; we did not expect to see such a large difference. We show our results in Figure 3.

	ASLR Disabled	ASLR Enabled	Percent Change
Debian	2963ns	2956ns	0.24%
FreeBSD	4368ns	4880ns	11.72%
HardenedBSD	3101ns	3133ns	1.03%

Figure 3: Performance test 2 results

3.3 Test 3

Our last two tests measure how long it takes to do some simple heap manipulations. The third test measures the amount of time it takes to map a single page into the virtual address space of the process using the $mmap$ system call. The mapping is set to be private and anonymous. This means that the mapping will not be shared with other processes and there is no underlying file for this mapping. The result of this mapping is a single page mapped into the address space placed at a random address at page granularity (meaning the randomization does not happen at any lower level than at the page level, as shown by the respective).

3.3.1 Test 3 results

This is the test that surprised us the most. There was a large slow down for Linux performance compared to the first 2 tests. More surprisingly, FreeBSD performed better with ASLR turned on for this test. We thought this was very strange, however we have performed this test 3 times and we have gotten similar results on every test. We are not sure if the performance decrease was caused by the kernel itself or the standard C library. We show our results for this test in Figure 4.

	ASLR Disabled	ASLR Enabled	Percent Change
Debian	630ns	644ns	2.22%
FreeBSD	1153ns	1073ns	-6.94%
HardenedBSD	913ns	938ns	2.74%

Figure 4: Performance test 3 results

3.4 Test 4

The final test measures how long it takes to do a word sized $malloc$. Because test 3 and 4 are so similar, we expect to see similar results here. This test will be affected by both the kernel and the standard library. Since calls to $malloc$ are extremely common, we hoped to see very small differences in performance.

3.4.1 Test 4 results

We were surprised to see a 3.74% decrease in Linux performance here. This is even larger than the number we calculated for test 3. We were even more surprised to see that HardenedBSD's $malloc$ ran faster when ASLR was enabled. This is most likely due to optimizations in the standard library compared to optimizations in the kernel. We show our results for this test in Figure 5.

	ASLR Disabled	ASLR Enabled	Percent Change
Debian	20301ns	21061ns	3.74%
FreeBSD	2948ns	2771ns	-5.98%
HardenedBSD	2110ns	2026ns	-3.98%

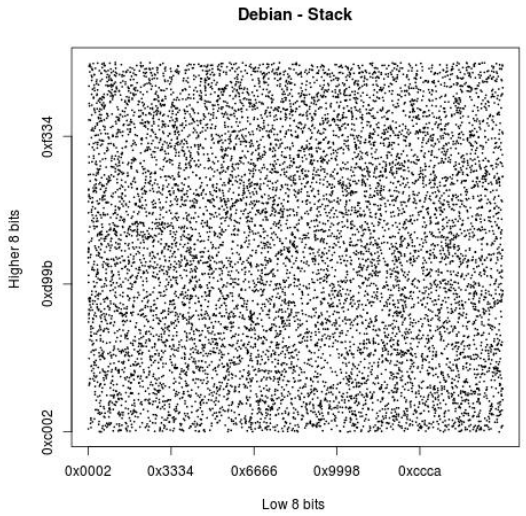
Figure 5: Performance test 4 results

4 Entropy Benchmarks

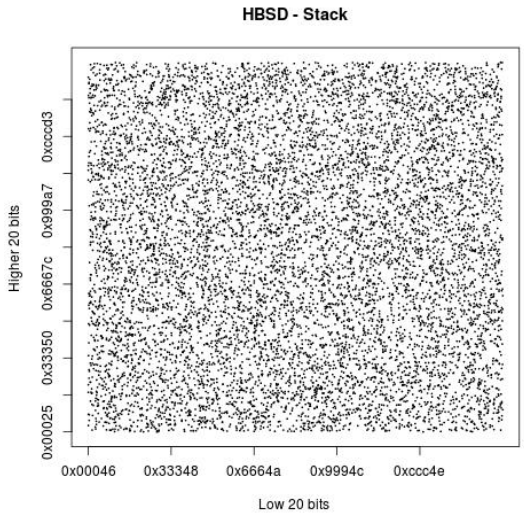
We designed four entropy experiments to measure the start address entropy within the different distributions. All of our experiments used a C program which simply did a $fork$ and $exec$ of the test we wished to run, for the number of times we wished to run it. Our graphs are 2-dimensional scatter plots in which the X axis is the lower bits of the address, and the Y axis is the higher bits of the address. In each test, we explain which bits of the address we use, and why we use them. In short, we truncated the addresses to just the bits of the address that are able to change. This stretched the graphs so that we are able to see the distribution more clearly.

4.1 Stack

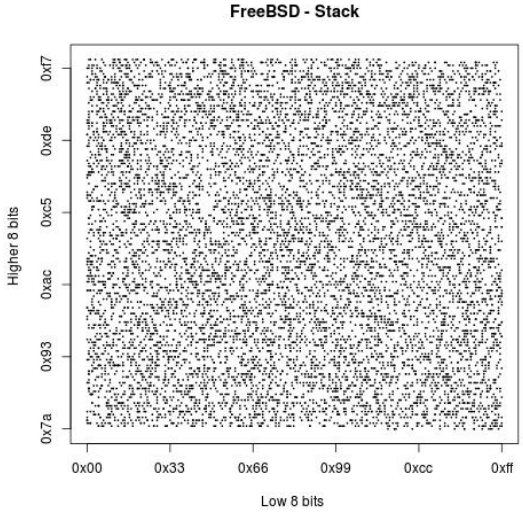
The first experiment measured the entropy of the stack. We accomplished this by printing the address of the argument list, effectively giving us the top of the stack. For Debian, we observed 30 bits of entropy in the stack. This was bits 4 to 34 (least significant to most significant). For FreeBSD, we observed 16 bits of stack entropy, bits 4 to 20. Since there is much less entropy here, we are able to observe the data as appearing more coarse in the plot in Figure 6c. Lastly, for HardenedBSD, we observed a surprisingly high 41 bits of entropy for the stack. This corresponded to bits 4 through 44 of the address. We show these results in Figure 6.



(a) Debian



(b) HardenedBSD



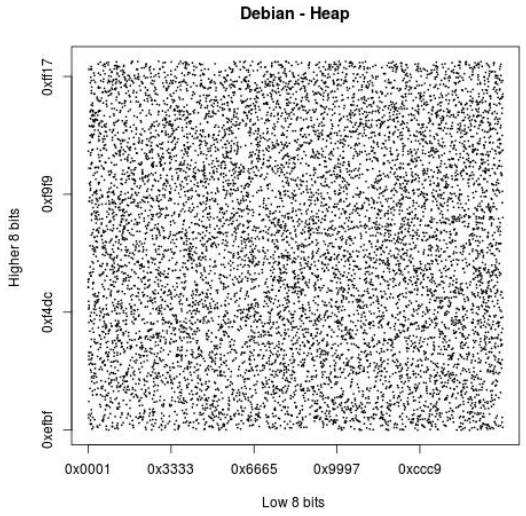
(c) FreeBSD

Figure 6: Stack entropy of Debian, HardenedBSD and FreeBSD

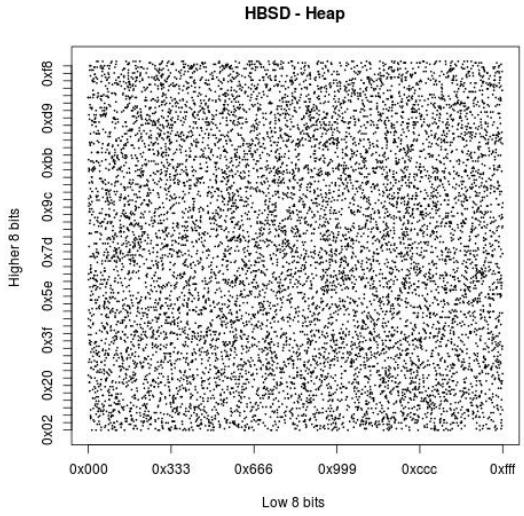
4.2 Heap

Our second experiment measured the entropy of the heap in a similar way. We created a variable and assigned to it the result of an $mmap$ call, then printed the address after the assignment. In Debian, we observe 29 bits of heap entropy, bits 12 through 41. In FreeBSD, we observe only 12 bits of entropy in bits 22 through 33. The plot for FreeBSD appears to be less uniform; this is because of the very low entropy. There aren't enough data points to evenly spread across the y axis. Because there are only 12 bits of entropy, there are only 4096 different possible stack addresses. We believe this randomization would do little to stop an attacker. Despite the FreeBSD ASLR being a patch from

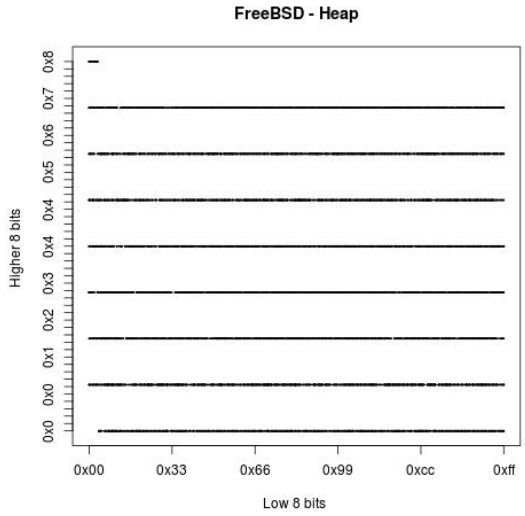
HardenedBSD, the entropy appears to be quite less. Lastly, for HardenedBSD, we observe 21 bits of entropy in bits 22 through 42. We show these results in Figure 7.



(a) Debian



(b) HardenedBSD



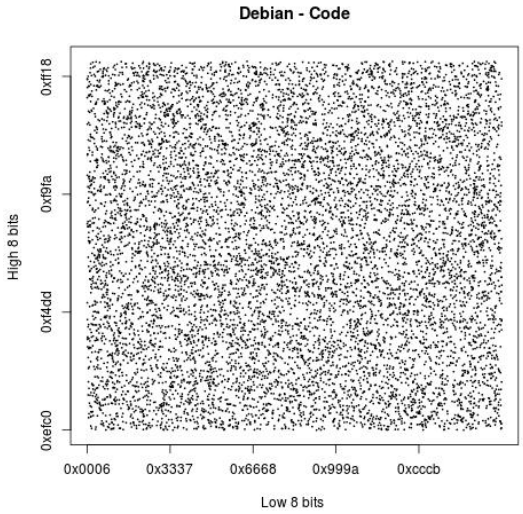
(c) FreeBSD

Figure 7: Heap entropy of Debian, HardenedBSD and FreeBSD

4.3 Code

Our third experiment, similarly, calculates the entropy of the code section of memory. To do this, we are printing the address of $main$ when the test is run. We were not able to get any FreeBSD and HardenedBSD did not have any randomization for the code section. The developers of HardenedBSD claim that there is a way to enable this, but we did not see any way to do this, despite exhaustively searching the documentation provided. Thus, we do not have

graphs or bits of entropy for the code entropy of HardenedBSD nor FreeBSD. For Debian, we observe 29 bits of code entropy, which is very good. These are bits 12 through 41, or nibbles 3 to 11. We show these results in Figure 8.



(a) Debian

Figure 8: Code entropy of Debian

.

4.4 Uniformity - χ^2

Visually, there does appear to be a relatively uniform distribution. However, we recognize that a simple visual test is not enough, and thus we ran a statistical test (Chi-Squared or χ^2) on all of our data. The reason we care about uniformity is because it is the main attack surface for ASLR. Thus, it is important that the memory addresses be uniformly distributed. Our results can be observed in Figure 9. We are skeptical of the validity of this test given the sparse nature of our data, but we provide the results (and the code for the tests in R in [B3]) nonetheless. We think that the FreeBSD results are accurate, however we believe that the tests failed for Debian and HardenedBSD because we did not supply enough data for the tests. We would have needed to generate billions of samples to get the χ^2 test to be able to pass.

	Stack	Heap	Code
Debian	Fail	Fail	Fail
FreeBSD	Pass	Fail	N/A
HardenedBSD	Fail	Fail	N/A

Figure 9: χ^2 test results

4.5 Bits of Entropy

Lastly, we calculated the bits of entropy observed in the starting addresses of the various memory areas. Although we mention the bits of entropy earlier, the tabular data is shown in Figure 10. As we mentioned earlier, FreeBSD and HardenedBSD do not have any entropy in the code section, and the table reflects this.

Operating System	Stack	Heap	Code
Debian	30 bits	29 bits	29 bits
FreeBSD	16 bits	12 bits	N/A
HardenedBSD	41 bits	21 bits	N/A

Figure 10: Bits of entropy observed

5 Utility of ASLR for Exploit Mitigation

Through our research we have found that ASLR is a very important layer of security to have in an operating system. ASLR is very effective on the modern x86_64 architecture because there are a lot more bits for randomization. We have shown that ASLR has a small performance cost. We believe that this cost is small enough to go unnoticed by a normal user. Compared to the zero bits of entropy in the memory start locations of an OS that does not use ASLR, the amount of entropy gained from ASLR is worth the minimal performance cost.

6 Limits of ASLR

ASLR is extremely effective on 64 addressable systems. However it is quite limited on 32 bit addressable systems, especially for page granularity allocations. When using $mmap$ on a 32 bit addressable system, given that pages are 4KB, this leaves a maximum of 20 bits of entropy for randomly placing the page. A full 20 bits of entropy is unlikely however, because the kernel usually maps itself into the top of the address space. 20 bits of entropy translates into just over one million possible addresses. This means that an attacker would have to run their exploit an average of five hundred thousand times before the exploit is successful. This is a trivial operation and can be done pretty quickly on 32 bit addressable hardware. The solution to this problem is to just move to 64 bit addressable hardware[A1]. Even with large pages, which are usually 4096KB, this allows a maximum of 42 bits of randomness which is over four trillion possible addresses. This is well outside the scope of probable exploitation.

7 Other Security Mechanisms

A secure system is usually not just equipped with one level of security, like ASLR. It usually also employs other mechanisms, such as Data Execution Prevention[A2], or policies like Jails in FreeBSD. These mechanisms usually do not help with some categories of vulnerabilities such as buffer over-reads. In March 2014, researchers found a catastrophic buffer over-read in OpenSSL, a commonly used SSL library used for web encryption[A4]. This vulnerability allowed an attacker to read 64KB of information from the heap, including currently stored usernames and passwords, of the running web server. We included some sample code in [B4] to exploit this vulnerability, which was known as Heartbleed or CVE-2014-0160. Buffer over-reads are in a class of vulnerabilities that are commonly unaffected by both ASLR and stack canaries. These types of vulnerabilities require other security mechanisms to help keep an attacker from being successful.

8 Conclusion

Overall, our findings and experiments supported our initial hypothesis that ASLR adds significant security with very minimal performance impact. The addition of ASLR helps slow down attackers at a minimal cost, and thus, it is definitely a feature that should be implemented in a secure operating system. FreeBSD's main branch currently does not have any support for ASLR. We believe that FreeBSD would gain a considerable amount of security by adding ASLR support. However, the current ASLR patch does not seem to provide quite the same level of entropy as HardenedBSD provides. We are not entirely sure why this is, but this may be something worth investigating in future work.

A References

[A1] Shacham, Hovav, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. ”On the Effectiveness of Address-space Randomization.” Proceedings of the 11th ACM Conference on Computer and Communications Security - CCS ’04 (2004): n. pag. Web.

[A2] Show, Kevin Z. ”IEEE Xplore Full-Text PDF:.” IEEE Xplore Full-Text PDF:. N.p., 2013. Web. 09 Dec. 2015.

[A3] Kil, Chongkyung. ”Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software.” IEEE Xplore. N.p., 2013. Web. 09 Dec. 2015.

[A4] Adrian, David. ”The Matter of Heartbleed.” ACM, 5 Nov. 2014. Web. 9 Dec. 2015.

B Appendix

[B1] O. Pinter, S. Webb. HardenedBSD ASLR patch for FreeBSD: https://github.com/HardenedBSD/hardenedBSD-upstreaming

[B2] R. Mutschlechner. Basic buffer overflow attack: https://gist.github.com/Ricky54326/54df69cb0a2c27595846

[B3] R. Mutschlechner, J. Detter. Code for experiments and graphing for this project: https://github.com/Ricky54326/CS736-ASLR-Benchmarks

[B4] R. Mutschlechner. Code for a Heartbleed exploit, in Python: https://gist.github.com/Ricky54326/cbbb7950a2e59aced732