ASLR: How Robust is the Randomness?

Jonathan Ganz
Department of Computer Science
University of California, Davis
jmganz@ucdavis.edu

Sean Peisert
Lawrence Berkeley National Laboratory
Department of Computer Science
University of California, Davis
speisert@ucdavis.edu

Abstract—**This paper examines the security provided by different implementations of Address Space Layout Randomization (ASLR). ASLR is a security mechanism that increases control-flow integrity by making it more difficult for an attacker to properly execute a buffer-overflow attack, even in systems with vulnerable software. The strength of ASLR lies in the randomness of the offsets it produces in memory layouts. We compare multiple operating systems, each compiled for two different hardware architectures, and measure the amount of entropy provided to a vulnerable application. Our paper is the first publication that we are aware of that quantitatively compares the entropy of different ASLR implementations. In addition, we provide a method for remotely assessing the efficacy of a particular security feature on systems that are otherwise unavailable for analysis, and highlight the need for independent evaluation of security mechanisms.**

I. Introduction

When configuring secure systems, it is important to consider the assurances that can be guaranteed by the mechanisms that enforce security. Buffer-overflow attacks are one of the most common classes of exploits against the control-flow integrity of computer systems. With increased reliance on embedded cyber-physical systems and software-defined networking, buffer-overflow attacks can damage much more than the individual system exploited [1], [2]. This threat is further complicated by the general reliance of organizations on third-party and closed-source software. Not only would the process of identifying and patching such vulnerabilities in source code be timely and costly, but the source code is not always available to patch.

A buffer-overflow vulnerability is a flaw in software written in memory-unsafe programming languages such as C [3]. These flaws occur when programs do not properly check the bounds on data that they write to memory [4]. The vulnerability becomes a bug when more data is written to memory than the amount of memory allocated for that data. Comparing the length of the data to be written with the length of the memory buffer allocated and throwing an exception, or otherwise handling the issue when the data will overflow the buffer, protects against this class of vulnerabilities. However, as with many error-handling issues in software engineering, bounds-checking is sometimes overlooked or handled incorrectly by programmers.

Address Space Layout Randomization (ASLR) is a class of computer security defense techniques designed to reduce the impact of buffer-overflow vulnerabilities. It adds a random offset to the virtual memory layout of each program, making it harder for an attacker to predict the target memory address that they wish the vulnerable program to return to. If the attacker overwrites the return address with a bad memory address, the probability of a successful exploit decreases.

In this paper, we evaluate the randomness that ASLR provides in different operating systems. We do this by quantitatively comparing the number of bits of entropy that each implementation produces and qualitatively comparing how the memory offset affects the return address over hundreds of executions of a vulnerable program. We note that there are other means of enhancing integrity [5], [6], [7] and other forms of measurement have been performed [8], [9], [10]. However, our focus in this paper is exclusively on an analysis of current ASLR implementations. Our contributions include a quantitative comparison of the effective entropy provided by different implementations of ASLR, and a technique for remotely evaluating the security of ASLR on computing systems that lack local access to the system or access to the operating system source code.

II. Related Work

The term *Address Space Layout Randomization* originally referred to a kernel patch for Linux developed by the PageEXec (PaX) Team, designed to protect against buffer-overflow attacks [11] and first released in 2001. ASLR quickly became a target for attackers interested in bypassing security [12], [13] and for researchers interested in improving the technology [14], [15].

Alternate methods for protecting the control-flow integrity on computers quickly followed. Some techniques aimed to add such security through a modified compiler [16] while others developed programs to automate the injection of memory protections into individual applications [17]. Though ASLR is the most well-known protection against buffer-overflow attacks, PaX is just one implementation of what has become a common kernel modification.

Microsoft first added ASLR to their Windows Vista operating system [18] and Apple's Mac OS X received an implementation of ASLR with version 10.5 in 2007 [19]. Since their initial releases, both operating systems have been updated with enhancements to their memory protection features.

ASLR is relied on by a vast array of systems, from corporate servers to mobile devices. Even Apple's iOS and Google's Android[1] mobile operating systems have implemented ASLR [20], though Android's low entropy has been found to be ineffective against all but the simplest of attacks [21]. With so many operating systems and therefore such a large portion of users relying on ASLR, it is important to investigate how well such implementations secure their environments.

ASLR has typically been implemented to protect against return-to-libc attacks [22]. With the advent and gradual adoption of ASLR in several operating systems, variations of these attacks were developed and can be categorized under the generic term of *return-oriented programming* (*ROP*) attacks [23] with different techniques adding various features [24], [25], [26]. Defenses from these enhanced attacks have also been developed [26], [27], but these solutions are often not as "turn-key" as ASLR, requiring developers to expend significant time and effort to obtain the benefits, so these security techniques are unlikely to become as widespread as ASLR.

Bittau et al. at Stanford developed an automated attack process for finding buffer-overflow vulnerabilities, even when ASLR is enabled. Their work on *Blind Return-Oriented Programming* (*BROP*) [28] demonstrates how even high-entropy ASLR-protected services, if not configured properly, can be attacked quickly and efficiently. The effectiveness of the BROP attack lies in its attack strategy: rather than brute-force the target's memory address, BROP discovers the current location in memory byte-by-byte, and from there, it can quickly search for useful ROP gadgets [24] to change the system's control flow. This method reduces the computational resources required to exploit a buffer-overflow vulnerability by orders of magnitude.

We perform an experiment which leverages this work, using the BROP attack's byte-by-byte technique for discovering vulnerable and hidden memory addresses. However, instead of measuring the number of ROP gadgets accessible from different programs or the number of requests required to exploit a buffer-overflow as Bittau does, we measure and map the bits that ASLR affects, giving us a quantitative value for the strength of each operating system's implementation of ASLR. When run hundreds of times, the attack program used in our experiments reveals the amount of entropy provided by the ASLR implementation of each operating system that is probed. Though this work measures the entropy of only BSD and Linux variants of ASLR, the program developed to measure entropy can be adapted to evaluate the effectiveness of ASLR implementations on other operating systems. It can also potentially be modified to measure alternative buffer-overflow mitigation techniques.

III. Experimental Procedure

The motivation for these experiments is that there may be some systems that can benefit from ASLR, but due to limitations in their processing power, architecture, licensing, or otherwise, they are incapable of compiling and running programs that are hardened to the recommended specifications [29]. We modify program compilation parameters to reflect such an environment, but first we describe how our program can be affected by a buffer-overflow.

Our vulnerable program, referred to as `server.c`, was developed to accept input from users through a network socket, copy it to a buffer of limited length, and then inform the user that the request has been serviced. The standard operation of this program is diagrammed below:

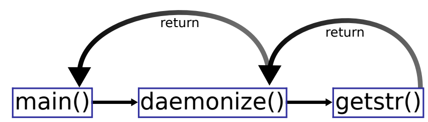

Fig. 1: Standard operation of the program `server.c`.

As the figure above illustrates, when the binary compiled from `server.c` executes, function daemonize() is called from main(), causing the process to fork for robustness against crashes. The program waits for a network message, at which point getstr() is called to copy the message into a buffer. Afterwards, getstr() returns execution to daemonize(), which promptly returns execution to main(). The infinite loop in main() continues to fork the process using daemonize() and wait for the next network request. However, due to the limited length of the buffer used in function getstr() and lack of bounds-checking, `server.c` contains a buffer-overflow vulnerability. If this vulnerability is exploited, the program's control flow can be manipulated, as in the following diagram:

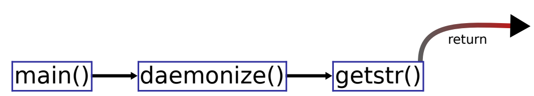

Fig. 2: Control flow of `server.c` during a buffer-overflow.

The program defined by `server.c` executes nominally, forking to service requests, until a string is submitted that exceeds the length of the buffer. After getstr() processes this string, one of three things can happen: the function can return execution to daemonize(), as is standard, it can attempt to return to an "invalid" address, causing the process to crash, or it can successfully return to another memory address, altering the program's control flow.

As alluded to above, we wish to produce an environment in which the security features of a device are limited. Therefore, the vulnerable program `server.c` was compiled with the following command to support basic ASLR defenses:

```
gcc -fPIE -pie -fno-stack-protector
    -O0 server.c -o server
```

The command above compiles the source code in `server.c` as a position-independent executable (parameters `-fPIE` and `-pie`) to take advantage of ASLR's features. Parameter `-fno-stack-protector` prevents the use of stack canaries that provide additional security against buffer-overflows [30]

[1]Security Enhancements in Android 1.5 through 4.1: https://source.android.com/security/enhancements/enhancements41.html

but may not be supported by specialized computer systems. Parameter `-O0` prevents the compiler from optimizing the hidden function away [31] and parameter `-o server` indicates that the executable file generated will be named server.

The figures in Section IV were generated by running an attack program `client.c`. The attack program targets the vulnerable server, sending it strings that incrementally approach the memory address of a hidden function. When this memory address is discovered by the attack program, it is logged. For each operating system analyzed, the attack program exploited the vulnerable server's buffer-overflow flaw over 700 times.

IV. Results

The following figures provide two types of visualizations to compare the randomness of memory addresses among several different operating systems. The first set of figures shows how each byte of memory changes with each run of the vulnerable server. The second set of figures provides a clear distinction between each byte of memory. It is important to show both visualizations because the first set of plots can reveal deterministic patterns over time whereas the second set allows for easier counting of the number of distinct values for each byte of memory. Knowing how many values a memory address can hold, as revealed by the second set of visuals, is not sufficient for determining its robustness against buffer-overflow attacks. The memory address can vary among trillions of values for a 64-bit operating system that allows ASLR to randomize 48 bits of the memory offset. However, if the memory address changes in a predictable way from one execution to the next, exploiting such a weakness would be simple.

Due to external constraints, we are unable to present the visualizations of all operating systems tested. However, we have tabulated the quantitative results, available in Section V, for all implementations of ASLR that we have evaluated for this work. The following figures display how the memory address of function `hidden()` in the vulnerable program varies with each execution, as discovered by our automated attack program.

Figure 3 below shows the memory layout over time of our vulnerable program when run on 32-bit OpenBSD. The most significant byte, represented by the orange circle, remains constant with a decimal value of 207 $(11001111)_2$. The second byte of the memory address, represented by the green triangle, varies among only five consecutive values in the upper half of the address space. Specifically, this byte varies among all values between 187 $(10111011)_2$ and 191 $(10111111)_2$. The third byte, represented by the blue square, varies greatly across the entire address space, but the least significant byte, shown as a purple diamond, varies among only sixteen constant values. Those values are listed below:

$$\begin{bmatrix} 0 & 16 & 32 & 48 & 64 & 80 & 96 & 112 & 128 \\ & 144 & 160 & 176 & 192 & 208 & 224 & 240 & \end{bmatrix}$$

We suspect that the lack of entropy in the least significant byte is due to paging requirements within all processes, including position-independent executables that support the features of ASLR.

Figure 4 shows the memory layout over time of our vulnerable program when run on 64-bit Debian Linux. The orange circle, the green triangle, and the blue square, representing the first, second, and third byte of memory, respectively for our target function, vary greatly across the entire address space. However, we observe again that the least significant byte of memory holds only sixteen unique values, distributed uniformly across the entire address space. The same sixteen values observed in 32-bit OpenBSD are observed in 64-bit Debian.

In figures 3 and 4, we plotted the value of each byte of memory over time, which can reveal patterns in the ASLR implementation's random number generator. Figures 5 and 6 visualize the same data in a different way. For each byte of memory, a dedicated column is used to plot every value observed by our attack program. Though any temporal pattern is lost, it is now easier to enumerate the values that each byte of memory can be assigned.

For the two operating systems that we have showcased, the least significant byte of memory varied among the same sixteen values. When viewed as decimal values, we observe that the least significant byte varies from 0 $(00000000)_2$ to 240 $(11110000)_2$ every 16 points. When represented in binary, we observe that the top four bits can hold any combination of 0 and 1, but the bottom four bits remain set at 0. Considering the binary representation of each byte can reveal the true limitations of each ASLR implementation.

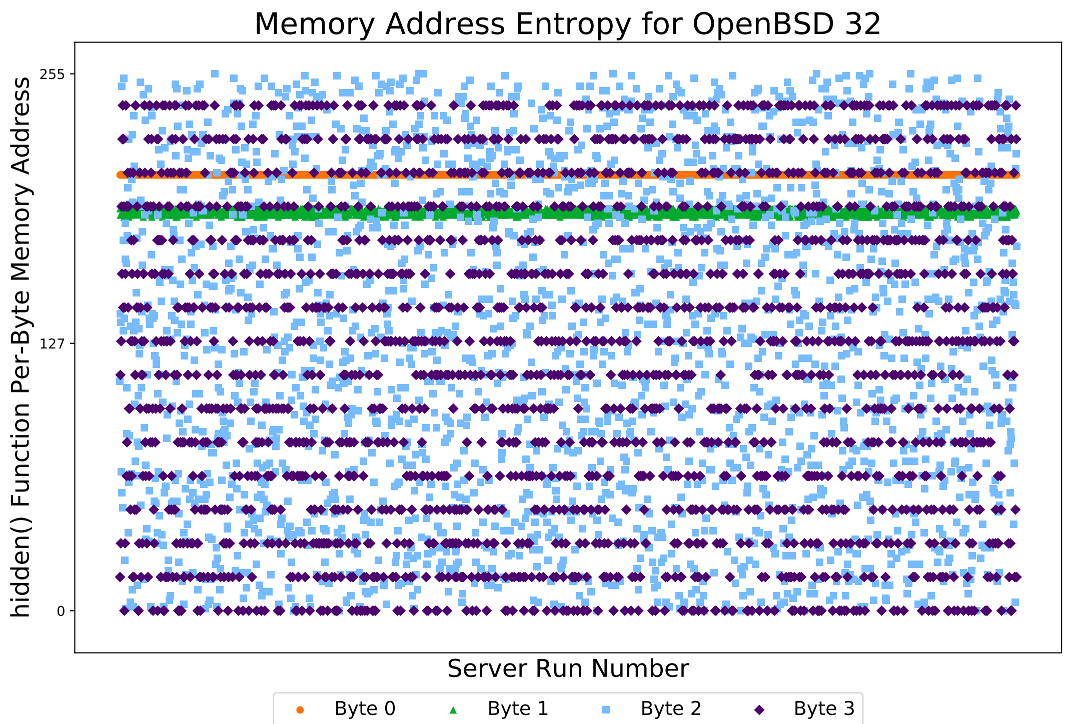

Fig. 3: Memory Address Layout in 32-bit OpenBSD.

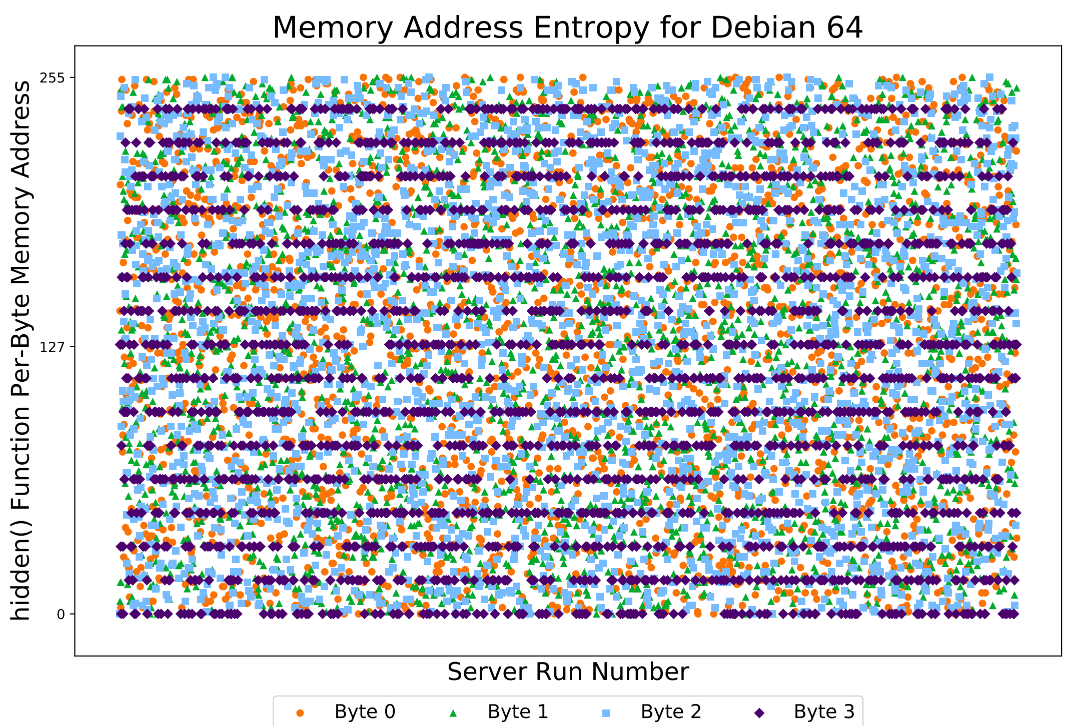

Fig. 4: Memory Address Layout in 64-bit Debian Linux.

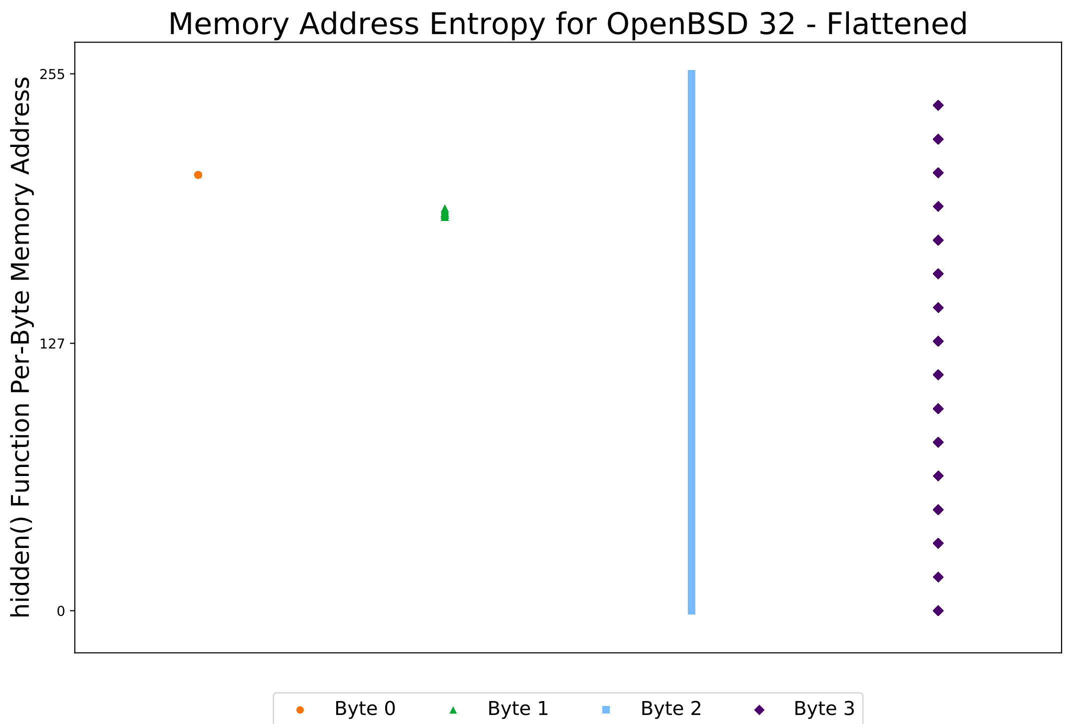

Fig. 5: Flattened Memory Address Layout in 32-bit OpenBSD.

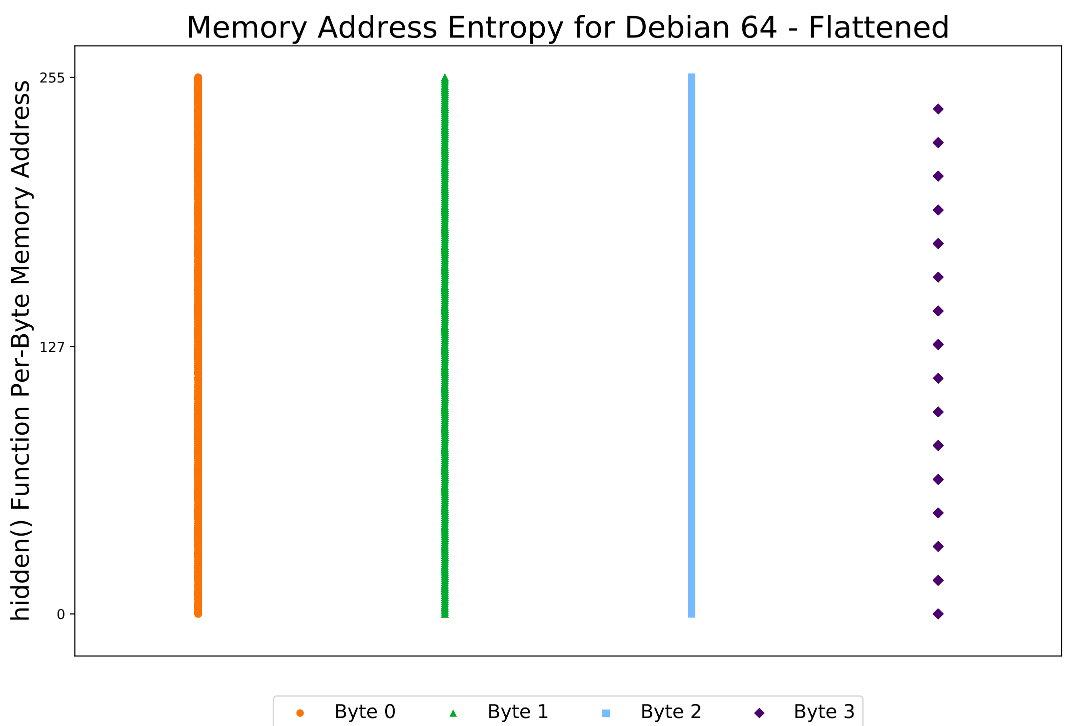

Fig. 6: Flattened Memory Address Layout in 64-bit Debian Linux.

V. Evaluation

Section IV provided visualizations of the security that each implementation of ASLR provides. In this section we provide the objective values for each operating system. The following table shows which bits of memory vary and the collective number of bits that can vary among each operating system tested.

Entropy of Each Operating System

Operating System	Changing Bits	Total Entropy
64-bit Debian	11111111111111111111111111110000	28 bits
64-bit HardenedBSD	00011111111111111111111111110000	25 bits
32-bit Debian	00000000111111111111111111110000	20 bits
64-bit OpenBSD	00000000000001111111111111110000	15 bits
32-bit OpenBSD	00000000000001111111111111110000	15 bits
32-bit HardenedBSD	00000000000000000000111111110000	8 bits

TABLE I: Comparison of ASLR implementations in various operating systems. The entropy and the specific bits of memory address affected are measured.

The table above lists the operating systems tested in order from the strongest to the weakest defense against buffer-overflow attacks. Also displayed are the specific bits that vary in separate runs of the vulnerable server. In each bitmap, bits designated 1 were observed to vary while bits designated 0 remained constant throughout the hundreds of attacks performed.

From a black-box testing perspective, the implementations of ASLR for 32-bit and 64-bit OpenBSD are identical. The same bits vary and remain constant in both operating systems. The figures generated by the data obtained in each OpenBSD environment are also remarkably similar. Although we observe that 64-bit operating systems generally have stronger defenses against buffer-overflow attacks, it is important to note that 64-bit OpenBSD is an exception to this trend. Its implementation of ASLR is weaker than that of 32-bit Debian. This is contrary to expectations, as 64-bit operating systems have more bits of memory address available to vary. The fact that 64-bit OpenBSD does not take advantage of its architectural advantage and provides less entropy than some 32-bit operating systems supports our argument for independent testing of security features.

64-bit Debian and 64-bit HardenedBSD had the two best implementations of ASLR, with 64-bit Debian having just slightly more entropy than 64-bit HardenedBSD. Although the values of entropy differ by only 3 bits, this corresponds to orders of magnitude greater robustness in the presence of buffer-overflow vulnerabilities. 32-bit Debian has a slightly weaker implementation of ASLR than its 64-bit counterpart: the most significant byte of memory does not vary between executions in 32-bit Debian.

Finally, 32-bit HardenedBSD, with an implementation of ASLR that provides only 8 bits of entropy, is the most vulnerable to buffer-overflow attacks. Only the bottom two bytes of memory varied during runs of the vulnerable server, indicating that it would take very few attempts to successfully exploit a buffer-overflow vulnerability in 32-bit HardenedBSD. Development of ASLR and hardware-dependent security mechanisms for 32-bit systems is considered a low priority for the developers of HardenedBSD, as most servers have transitioned to 64-bit platforms according to download statistics.

Each operating system tested has its own implementation of ASLR. The differences of each implementation are evident when comparing all operating systems of equivalent hardware architectures: all 64-bit operating systems have a different set of varying bits; similarly, all 32-bit operating systems have a different set of varying bits. However, an in-depth analysis of the cause for differences between each implementation is outside the scope of this work.

VI. Discussion

In general, when examining the memory layout of computer programs, we expect to see patterns related to powers of 2: the binary configuration of computers dictates this. However, in running the experiments above, some bytes of memory exhibited variation among a number of values that were slightly off from the power of 2 that we should observe. On 32-bit HardenedBSD, the third byte of memory took on 9 unique values. On 32-bit OpenBSD, the second byte of memory took on 5 unique values.

Could these anomalies be due to bit-flipping? This seems unlikely, as the least common value taken on by each byte corresponded to 2.64% and 4.41% of all values observed, respectively. Though less common than the other values, this frequency is far too high to be explained by a bit-flipping error. The more likely explanation is that unique properties of the pseudorandom number generator (PRNG), or of the ASLR code that uses the PRNG's output, results in unexpected values of memory used and certain numbers being more likely to be observed than others.

VII. Limitations

In performing this research, there were factors that prevented ideal analysis of each ASLR implementation's entropy. This section documents those limitations, how they might affect our results, and what we did to minimize any negative impact.

These experiments examined only variants of BSD and Linux. It would be useful to expand the study to variants of Apple's macOS and Microsoft's Windows operating systems. Comparing ASLR in more variants provides a better understanding of which operating systems best resist buffer-overflow attacks.

Our work focused on black-box penetration testing of these operating systems with almost no investigation into the differences in the source code that may be responsible for the differences in entropy that is observed. Though some operating systems such as macOS and Windows do not provide the source code required to adequately investigate the cause of these differences, this can be done for all operating systems that have already had their ASLR implementation profiled.

Initially, we had attempted to use a pre-existing piece of software with a well-known buffer-overflow vulnerability, documented as a CVE.[2] However, there were challenges in obtaining old variants of such software for each operating system that we chose to investigate. For compatibility reasons and so that the comparisons being made are with the same software, we decided to develop our own vulnerable software. This allowed us to ensure that the vulnerability being measured is the same across all platforms and it gives us a better understanding of the source code running on both the attacker and the victim system.

VIII. Future Work

We hope to enhance this work by measuring the effective entropy of the ASLR implementations in other mainstream operating systems, particularly versions of Apple's macOS and Microsoft's Windows. Google's Android mobile operating system would be another interesting platform to compare. Due to the closed nature of Apple's iOS environment, it may be difficult to run our measurement software on that operating system.

As mentioned in Section VII, performing source code analysis would be useful to determine which differences in each ASLR implementation are responsible for the differences in entropy that we observe. Such analysis is beyond the scope of this current research, but would certainly provide greater insight into which techniques provide the greatest entropy and how to augment the randomness in weaker variants.

IX. Conclusion

This work quantitatively compares the ability of Address Space Layout Randomization implementations to defend against buffer-overflow attacks. We rank the security of operating systems and their architecture based on the amount of entropy provided by their ASLR implementation. This is measured by running a vulnerable program and attacking it multiple times, recording each memory address that results in successful manipulation of the program's control-flow. By performing hundreds of measurements on the random memory address assigned to a specific target function, we are able to determine the amount of effective entropy observed in various implementations of Address Space Layout Randomization.

We find that among the operating systems profiled, 64-bit Debian Linux has the strongest defense against buffer-overflow attacks while 32-bit HardenedBSD has the weakest defense. In the case of OpenBSD, using the 64-bit variant provides no additional security for this attack scenario.

Although there are myriad choices for desktop operating systems, several of which have adequate security mechanisms, the realm of embedded systems offers far fewer options as far as security goes. With the rapid increase in popularity of Internet-of-Things devices, one may be wary of the compromises made in terms of security in order to achieve increased compatibility and power efficiency. It is our hope that this analysis can be a first step in evaluating the control-flow integrity of such embedded operating systems.

[2] Common Vulnerabilities and Exposures: https://cve.mitre.org/cve

X. Acknowledgements

This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsors of this work.

References

[1] P. Koopman, "Embedded System Security," *Computer*, vol. 37, no. 7, pp. 95–97, 2004.

[2] D. Midi, M. Payer, and E. Bertino, "Memory Safety for Embedded Devices with nesCheck," in *Proceedings of the 12th ACM Asia Conference on Computer and Communications Security*, pp. 127–139, ACM, 2017.

[3] G. C. Necula and P. Lee, "The Design and Implementation of a Certifying Compiler," *ACM SIGPLAN Notices*, vol. 33, no. 5, pp. 333–344, 1998.

[4] A. One, "Smashing the Stack for Fun and Profit," *Phrack*, vol. 7, no. 49, 1996.

[5] J. P. McGregor, D. K. Karig, Z. Shi, and R. B. Lee, "A Processor Architecture Defense Against Buffer Overflow Attacks," in *Proceedings of the 1st International Conference on Information Technology: Research and Education*, pp. 243–250, IEEE, 2003.

[6] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein, "seL4 Enforces Integrity," in *Proceedings of the 2nd International Conference on Interactive Theorem Proving*, pp. 325–340, Springer, 2011.

[7] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "seL4: From General Purpose to a Proof of Information Flow Enforcement," in *Proceedings of the 34th IEEE Symposium on Security and Privacy*, pp. 415–429, IEEE, 2013.

[8] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn, "Design and Implementation of a TCG-based Integrity Measurement Architecture," in *Proceedings of the 13th USENIX Security Symposium*, pp. 223–238, 2004.

[9] D. Muthukumaran, A. Sawani, J. Schiffman, B. M. Jung, and T. Jaeger, "Measuring Integrity on Mobile Phone Systems," in *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*, pp. 155–164, ACM, 2008.

[10] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-Flow Integrity: Precision, Security, and Performance," *ACM Computing Surveys*, vol. 50, no. 1, pp. 16:1–16:33, 2017.

[11] P. Team, "PaX Address Space Layout Randomization (ASLR)," *Documentation for the PaX Project*, 2003. Available at https://pax.grsecurity.net/docs/aslr.txt.

[12] T. Durden, "Bypassing PaX ASLR Protection," *Phrack*, vol. 59, no. 9, p. 9, 2002.

[13] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, *et al.*, "Address-Oblivious Code Reuse: On the Effectiveness of Leakage-Resilient Diversity," in *Proceedings of the 24th Network and Distributed System Security Symposium*, 2017.

[14] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the Effectiveness of Address-Space Randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pp. 298–307, ACM, 2004.

[15] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi, "Leakage-resilient layout randomization for mobile devices," in *Proceedings of the 23rd Network and Distributed System Security Symposium*, 2016.

[16] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard TM: Protecting Pointers from Buffer Overflow Vulnerabilities," in *Proceedings of the 12th USENIX Security Symposium*, vol. 12, pp. 91–104, 2003.

[17] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits," in *Proceedings of the 12th USENIX Security Symposium*, pp. 105–120, 2003.

[18] O. Whitehouse, "An Analysis of Address Space Layout Randomization on Windows Vista," *Symantec Advanced Threat Research*, pp. 1–14, 2007.

[19] A. Keromytis, "Randomized Instruction Sets and Runtime Environments," *Proceedings of the 30th IEEE Symposium on Security and Privacy*, pp. 18–25, 2009.

[20] C. Miller, "Mobile Attacks and Defense," *Proceedings of the 32nd IEEE Symposium on Security and Privacy*, vol. 9, no. 4, pp. 68–70, 2011.

[21] S. Liebergeld and M. Lange, "Android Security, Pitfalls and Lessons Learned," in *Proceedings of the 47th Annual Conference on Information Sciences and Systems*, pp. 409–417, IEEE, 2013.

[22] 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*, pp. 552–561, ACM, 2007.

[23] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pp. 27–38, ACM, 2008.

[24] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-Oriented Programming Without Returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pp. 559–572, ACM, 2010.

[25] 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*, pp. 30–40, ACM, 2011.

[26] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pp. 40–51, ACM, 2011.

[27] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free: Defeating Return-Oriented Programming Through Gadget-Less Binaries," in *Proceedings of the 26th Annual Computer Security Applications Conference*, pp. 49–58, ACM, 2010.

[28] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, "Hacking Blind," in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, pp. 227–242, IEEE, 2014.

[29] M. Payer, A. Barresi, and T. R. Gross, "Fine-Grained Control-Flow Integrity through Binary Hardening," in *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 144–164, SIG SIDAR, 2015.

[30] P. Wagle, C. Cowan, *et al.*, "Stackguard: Simple Stack Smash Protection for GCC," in *Proceedings of the GCC Developers Summit*, pp. 243–255, GNU Project, 2003.

[31] R. M. Stallman and the GCC Developer Community, *Using GCC: The GNU Compiler Collection Reference Manual*. GNU Press Boston, 2003.