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 [11], [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 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 mobile operating systems have implemented ASLR, though Android’s low entropy has been found to be ineffective against all but the simplest of attacks. 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. 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 with different techniques adding various features. These advanced attacks have also been developed, 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) 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 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. 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, 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 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 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 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 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 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. 

---

1 Security Enhancements in Android 1.5 through 4.1: [https://source.android.com/security/](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 \(-o\) \texttt{server} indicates that the executable file generated will be named \texttt{server}.

The figures in Section [IV] were generated by running an attack program \texttt{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 \texttt{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 \((11001111)\). 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 \((10111011)\) and 191 \((10111111)\). 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 \((00000000)\) to 240 \((11110000)\) 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.
TABLE I: Comparison of ASLR implementations in various operating systems. The entropy and the specific bits of memory address affected are measured.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Changing Bits</th>
<th>Total Entropy</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit Debian</td>
<td>1111111111111111111111110000</td>
<td>28 bits</td>
</tr>
<tr>
<td>64-bit HardenedBSD</td>
<td>0011111111111111111111110000</td>
<td>25 bits</td>
</tr>
<tr>
<td>32-bit Debian</td>
<td>0000000011111111111111110000</td>
<td>20 bits</td>
</tr>
<tr>
<td>64-bit OpenBSD</td>
<td>0000000000011111111111110000</td>
<td>15 bits</td>
</tr>
<tr>
<td>32-bit OpenBSD</td>
<td>0000000000000111111111110000</td>
<td>15 bits</td>
</tr>
<tr>
<td>32-bit HardenedBSD</td>
<td>000000000000000011111111110000</td>
<td>8 bits</td>
</tr>
</tbody>
</table>

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.
Initially, we had attempted to use a pre-existing piece of software with a well-known buffer-overflow vulnerability, documented as a CVE. 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.

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