Effective Entropy: Security-Centric Metric for Memory Randomization Techniques

William Herlands
MIT Lincoln Laboratory

Thomas Hobson
MIT Lincoln Laboratory

Paula J. Donovan
MIT Lincoln Laboratory

Abstract

User space memory randomization techniques are an emerging field of cyber defensive technology which attempts to protect computing systems by randomizing the layout of memory. Quantitative metrics are needed to evaluate their effectiveness at securing systems against modern adversaries and to compare between randomization technologies. We introduce Effective Entropy, a measure of entropy in user space memory which quantitatively considers an adversary's ability to leverage low entropy regions of memory via absolute and dynamic inter-section connections. Effective Entropy is indicative of adversary workload and enables comparison between different randomization techniques. Using Effective Entropy, we present a comparison of static Address Space Layout Randomization (ASLR), Position Independent Executable (PIE) ASLR, and a theoretical fine grain randomization technique.

1 Introduction

Memory based exploits take advantage of software vulnerabilities, allowing an adversary to hijack control flow of a process and execute arbitrary code. While advancements in security techniques have eliminated the simplest forms of code injection attacks [22], more sophisticated attacks continue to defeat modern security systems. Contemporary adversaries boast a substantial repertoire of techniques to overcome security features including heap spraying [8], integer overflow [23], and return oriented programming [18] [5] (ROP), a generalization of the earlier return to libc schemes [25].

User space memory randomization is an active sector of security research which attempts to protect a computing system by randomizing its memory layout. These techniques are premised on the assumption that adversaries often require precise knowledge of the location of sections, functions, and instructions in memory. Therefore, randomizing the locations of these elements will increase the difficulty to exploit a system. Yet these techniques lack quantitative metrics to evaluate their effectiveness in protecting memory against contemporary attacks and how they affect adversary workload. In this paper we introduce Effective Entropy (EffH), a novel metric to quantifiably assess the security of memory randomization technologies based on specific threat models.

2 Related Work

In response to the proliferation of memory-based exploits, researchers have developed a litany of security protections for user space memory. Two widely implemented techniques include concealed memory address values called stack canaries, which detect code injection attacks [6] [9], and W$\oplus$X, which prevents memory pages from being simultaneously writable and executable.

Recent literature has highlighted user space memory randomization schemes. Address Space Layout Randomization (ASLR) is the most commonly deployed randomization technology. Versions of ASLR are standard in the most recent versions of Mac OS X, Windows, Linux, and FreeBSD as well as a number of smartphone operating systems. For example, under standard conditions in Linux ASLR implementations, the kernel facilitates load time randomization of an arbitrary process's virtual memory. Specifically, the base address of the stack, heap, virtual dynamically linked shared objects (VDSO), and memory mapped (`mmap`) sections are each randomized, though with different amounts of entropy [4]. In contrast, Position Independent Executable (PIE) ASLR additionally randomizes the base address of the program image (including the Text, Read-Only Data, and Read-Write Data sections) as a single block by appending the program image to the randomized `mmap` section. In Linux distributions static (non-PIE) ASLR is standard and a program can be specifically compiled with a PIE option. However, PIE is standard in recent OpenBSD distributions [7] which is indicative of a trend towards more hardened memory space security in commodity operating systems.

Research on fortifying memory randomization has generated a variety of proposals which focus on all sections of user space memory. While many of these security enhancements combine a number of elements, they mainly focus on either base address or fine-grain randomization security techniques. For example, proposals such as Address Space Layout Permutation [14] increase the magnitude of entropy for the base addresses above contemporary ASLR implementations. In order to achieve greater randomization, these techniques leverage the fact that the size of memory for a process is small with respect to the size of memory potentially available to the user. Alternatively, fine-grain randomization technologies such as Binary Stirring [24], which provides randomization to the Text section, attempt to increase the granularity of randomization to resolutions smaller than memory sections. Similarly, Multicompiler [12], which also provides randomization within the Text section, randomly adds NOPs between blocks of code. In addition to base and fine-grain randomization, a number of proposals attempt to dynamically re-randomize sections in memory during the execution of the program. For example, the Minix Kernel enables dynamic randomization of base addresses in operating system processes [10] and Instruction Location Randomization has the capability for dynamic fine grain randomization in the Text section [11].

Each of these techniques has specific requirements for the source code or binaries it requires in order to operate properly. For the purposes of this paper, we are not concerned with these specifications, although widespread adoption of any system certainly favors the most generally applicable technology. Throughout this paper, unless otherwise noted, we focus on Ubuntu 12.04 machines and its static ASLR and PIE ASLR implementations.

Other security randomization proposals, such as Instruction Set Randomization [13], In Place Code Randomization [15], and Data Space Randomization [1] also propose protecting a computer system by obfuscating aspects of a process' memory from the attacker. However, these schemes do not randomize the layout of memory and thus fall outside the scope of our evaluation.

3 Threat Models

As formalized in Section 5, EffH is dependent on the specifics of the threat model of an adversary. This specificity enables EffH to provide an accurate assessment of the security impact of a memory randomization technology. Below we describe two adversarial threat models over which we will evaluate security techniques: the Advanced Adversary and the Memory Disclosure Adversary. These threats are commensurate with the skill and versatility of modern adversaries who are able to leverage multiple vulnerabilities to launch successful attacks.

The Advanced Adversary is a sophisticated adversary who desires to execute arbitrary code on a target machine and has knowledge of a zero day vulnerability in an arbitrary program. Additionally, the adversary has access to a binary identical to the one being run on the target machine. The vulnerability allows the adversary to write to a writable section of memory in an attempt to take control of the instruction pointer. In order to isolate the effects of memory randomization, we assume the adversary can circumvent all other security features (with the notable exception of W$\oplus$X protection) without learning anything which would affect the entropy of memory.

The Memory Disclosure Adversary is similar to the Advanced Adversary but also has partial knowledge of the layout of memory obtained through an information disclosure attack. Such attacks force the machine to disclose an obfuscated address or permit the adversary to read a controlled area in memory, and are a common means of reducing the entropy in an arbitrary section of memory [17]. A complete consideration of the difficulty of using an information disclosure to reduce entropy is beyond the scope of this paper. Rather, we consider a simple but practical Memory Disclosure Adversary that can disclose the location of one section of memory in its entirety. A lack of intra-section randomization in today's systems makes this type of disclosure practical; objects have fixed relative positions and an adversary that discloses one address in a section has effectively disclosed the location of every other object in that section. Under this adversarial model we evaluate randomization techniques in terms of the effect that a complete disclosure of one section has on the other sections.

4 Security Metrics

Two of the most commonly used metrics for evaluating memory randomization techniques (a) measure the entropy introduced into a system and (b) test systems against known exploits.

4.1 Entropy

Entropy, H, defined in equation 1, is a quantitative and information theoretic measurement which measures the unpredictability of a random variable, X, over its possible states, $x_i \in X$. Measuring the individual entropy of each section in memory is fairly straightforward. Additionally, measures of entropy are easily compared between memory sections and randomization technologies. Given the ease of measurement and objectivity of results, this is the standard means by which memory randomization techniques are currently evaluated.

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log p(x_i) \tag{1}$$

Utilizing the entropy provided to each section as a measure of security assumes that greater entropy correlates with great difficulty on behalf of the adversary attempting exploit a system. Yet this assumption does not hold under the modern adversary threat models discussed in Section 3. These adversaries are able to launch attacks which circumvent the entropy of various memory sections, potentially rendering even high entropy useless. Indeed, two of the most significant papers on evaluating memory randomization security demonstrated that using specially crafted memory attacks, they could avoid the entropy in static ASLR [19] and a fine grain ASLR implementation [20]. Their evaluations demonstrated fundamental weaknesses in the presumed relationship between entropy and security.

4.2 Exploits

Since randomization techniques attempt to prevent adversaries from exploiting a process running on a target machine, it seems intuitive to measure the security by directly testing the technology against potential attacks. More specifically, launching a set of known exploits against an unprotected system and against a system protected by the randomization technology would demonstrate which attacks were prevented by the additional security. Furthermore, specially crafted attacks can demonstrate weaknesses within a security system and perhaps challenge some of the underlying assumptions of that technology.

However, analysis of individual exploits is fundamentally limited in the information it can provide about a security system since exploits do not provide a quantitative assessment of computing system. Additionally, it is difficult to compare different exploits, or make comparisons between heterogeneous security systems using the same exploits. Finally, since exploits are developed to take advantage of specific nuances of a system, trivial changes in a system (such as dynamically switching between endianness) can break specific implementations of current exploits without fundamentally increasing security.

4.3 New Metrics

In order to properly evaluate the security offered by diverse memory randomization technologies, metrics must be developed which can quantifiably access a large set of randomization schemes. Learning from the advantages and drawback of using entropy and exploits as metrics, we can identify some necessary qualities in evaluations:

1. Quantitative, not anecdotal
2. Comparable between existing security systems and future technologies
3. Objective, unbiased towards specific exploits
4. Predictive of adversary workload

5 Effective Entropy

EffH attempts to provide meaningful measurements of effectiveness by combining the quantitative rigor of entropy with domain knowledge regarding the capabilities of modern adversaries. Specifically, EffH considers an attacker's capability to circumvent entropy in memory by leveraging interconnections within memory.

5.1 Memory Connections

Although memory is divided into a number of sections, these individual sections interact with each other throughout the lifetime of a process. Interconnections between sections in user space memory can either be absolute or dynamic. Absolute connections exist throughout the execution of a program regardless of the state of the process. Dynamic connections form and break over the course of a program's execution and exist only during certain states of a process. Table 1 shows examples of the connections in memory.

Absolute connections include connections formed by jump and call instructions which move the instruction pointer to a fixed address as well as pointers located in read-only memory. Dynamic connections are formed by the values of registers and writable pointers at every moment where an attacker could gain control of the program's execution.

These register and pointer values could be used by attackers leveraging gadgets with indirect branch instructions. Given the computational power of even relatively few gadgets [16] we assume that any executable section has the necessary instructions to leverage the values in registers. Similarly, attackers can leverage pointers in memory, though the attackers may have to overcome additional entropy, as discussed below.

5.2 Definition

The importance of memory connections can be understood through a simple example. Consider a static ASLR protected system, where every instruction in the program image connects to an address in the `mmap`. Although the `mmap` is randomized with 8 bits of entropy, ASLR does not provide the program image with any entropy. An adversary can use the connections between the program

Absolute	**Dynamic**
`jmp 0x8057420`	`jmp *0xBF04084`
	`jmp %ebx`
Read-only pointer	Writable pointer

Table 1: Examples of absolute and dynamic connections in user space memory.

image and `mmap` to reach every address in the `mmap` with impunity. Thus, the `mmap` effectively has zero bits of entropy. Furthermore since only the base address of the `mmap` is randomized, a single connection between the program image and `mmap` is sufficient for an adversary to know the location of all addresses in the `mmap` with zero bits of entropy. This entropy reduction technique is not novel, and is routinely employed in attacks against static ASLR [21].

EffH measures the effective magnitude of entropy in each memory section that an adversary must directly confront and cannot circumvent by taking advantage of inter-memory connections. Since the ability for an adversary to leverage inter-memory connections is dependent on their threat model, EffH models must be constructed to specifically address each threat model.

Equation 2 defines EffH of each section, s, for both the Advanced Adversary and Memory Disclosure Adversary threat models. Connections from executable sections reduce the initial entropy of the target section, h_s, to the minimum entropy of any executable section with a connecting instruction, $min(H^x_{conn})$ if $min(H^x_{conn}) < h_s$. In order to leverage pointers in writable or RO memory these adversaries must use gadgets from an executable section, as discussed above. Therefore connections from pointers reduce $EffH_s$ to the sum of the minimum entropy of any section with a connecting pointer, $min(H^p_{conn})$, and the minimum entropy of all executable sections $min(H^x)$.

$$EffH_s = min \begin{cases} h_s \\ min(H^x_{conn}) \\ min(H^p_{conn}) + min(H^x) \end{cases} \quad (2)$$

$$H^p_{conn} = \{h^p_j : \exists connection(j,s)\}$$

$$H^x_{conn} = \{h^x_j : \exists connection(j,s)\}$$

EffH better addresses realistic threat models than standard entropy because EffH accounts for the ability of an attacker to use connections to circumvent the entropy of various sections. Faced with uncertainty about the location of a section in memory an attacker may (a) repeatedly probe the process to probabilistically reduce entropy (b) obtain a memory disclosure to eliminate entropy (c) accept a lower rate of success inversely proportional to the EffH of the desired section. Since all of these options require more effort on behalf of the attacker or reduce the chance of a system being exploited, EffH measures the increased adversary workload.

Considering the necessary characteristics for robust metrics defined in Section 4, the values of EffH for each memory section are both quantitative and readily comparable between different systems. Additionally, EffH is not dependent on a particular threat model. Finally, EffH is indicative of difficulty faced by an adversary because EffH represents the true randomness that an adversary confronts in memory.

Consider the example shown in Figure 1 of a static (non-PIE) 64-bit Linux ASLR implementation that randomizes the Stack section with 22 bits of entropy, Memory Maps (mmap) with 28 bits, and the Heap with 13 bits. The Program Image itself is not randomized (0 bits of entropy). We assume that an Advanced Adversary is able to exploit a stack buffer overflow to write arbitrary values to the stack, including overwriting the return address.

An attacker wishing to call the *exec()* function might attempt to directly guess the address of *exec()* by overwriting the return address with a guessed address. However, the attacker would confront the full 28 bits of entropy of the mmap section. A more fruitful approach would be to leverage the existing *exec_ptr* in the heap, which points to the *exec()* function. The attacker cannot directly overwrite the return address with a guess of the location of *exec_ptr* due to the layer of indirection associated with using a pointer stored on the heap. To use *exec_ptr* indirectly the attacker may use gadgets in any sufficiently large executable section. In this case the attacker chooses a gadget from the Program Image, as it is not randomized.

5.3 Example

The attack unfolds as follows. The attacker overwrites the return address on the stack with the address of the gadget in the Program Image (1). The attacker also overwrites the next value on the stack with a guess for the address of *exec_ptr* (2). When the program returns, the invoked gadget in the Program Image will pop the attacker's guess of *exec_ptr* off the stack and jump to the value contained at that address. If the attacker's guess of *exec_ptr* is successful, control flow will be redirected to the *exec()* function (3).

To calculate $EffH_{mmap}$ we use Equation 2. The first component of Equation 2, h_s, is simply the entropy of the mmap section, thus $h_{mmap} = 28$. The next component, $min(H^x_{conn})$, calculates the entropy of any absolute references from executable sections to the mmap section; for this example we assume there are none. Finally, we must calculate $min(H^p_{conn})$ + $min(H^x)$, the minimum entropy of any sections containing pointers to the

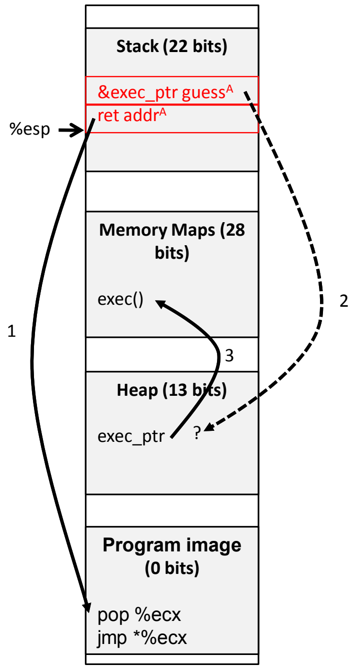

Figure 1: EffH of Memory Maps reduced to 13 bits due to pointer from Heap (13 bits) that may be accessed via gadgets in the Program Image (0 bits). 'A' denotes attacker supplied values

mmap section (i.e. the Heap) plus the minimum entropy of any sections containing executable code (i.e. the Program Image), which contain gadgets that allow us to use the pointer on the heap. In this example, $min(H^{p}_{conn}) + min(H^{x}) = min(h_{heap}) + min(h_{program\_image}) = 13 + 0$. Thus, $EffH_{mmap} = min(28, 13) = 13$.

6 Methodology

In order to measure EffH for a particular randomization technology, we empirically calculated the entropy of all memory sections by running a sample program ten million times and sampling the base address of every randomized section during each run.

To determine the absolute connections static analysis was used on sample programs to identify all the intended and unintended fixed control instruction connections and pointers in non-writable sections of memory.

To determine the dynamic connections in the sample programs we used an algorithmic approach similar in concept to conservative garbage collectors [3]. All indirect branch instructions which use pointers in writable memory (such as jmp *<W address>) were identified. The sample programs were each run twice through identical, deterministic execution paths. During those runs the memory of the process was examined at each point where an adversary could take control of the program (i.e. at every control transfer instruction). At those instances, we stored the values of registers as well as intended and unintended pointers in writable memory. In this manner we identified every potential dynamic connection formed during each run. In our evaluation we assumed that under both the Advanced and Memory Disclosure Adversary models that dynamic connections existing at any control transfer instruction could be used by the adversaries during the actual control transfer instruction that is used to hijack control flow.

A potential dynamic connection is any value in a register or writable memory which evaluates to an address allocated in user memory space. Since memory sections are randomized for each run, false dynamic connections can be registered if a connection target value happens to equal that of valid user space memory. For example, an `int` value in writable memory could be incorrectly identified as a pointer. In the post-processing step, false connections were eliminated by taking the intersecting set of dynamic connections from the two independent runs on each program, each of which were executed on independently randomized systems.

The rate of false connections was quite low, usually around 1.5% of potential dynamic connections were false. Since under normal conditions, user space memory on Linux 32-bit systems begins at memory address 0x8048000, it is reasonable that very few false connections are formed by accidental values in writable memory. While these algorithms do not directly account for false connections to sections in memory with no entropy, these false connections do not affect EffH since they terminate in sections with already minimal entropy.

In Section 7 we discuss measurements of ROP gadgets and system calls in executable sections. We determined the number of ROP gadgets by statically counting all series of instructions ending with a control flow instruction. We used an every munch algorithm with a maximal instruction length of 5 [16] and considered both intended and unintended gadgets [2]. We searched for system calls by similarly evaluating instructions beginning at every address in executable memory.

Although EffH is measured on a per-program level, the metric itself is a function of a randomization technology and a threat model. Our ability to generalize the results of per-program measurements requires two assumptions regarding program execution flow and memory connections.

1. Any sufficiently large program is assumed to have relevantly similar sets of connections between memory sections. An example of insufficiently large program is one which does not utilize a section in memory, such as the heap, or one with a degenerately small number of instructions.

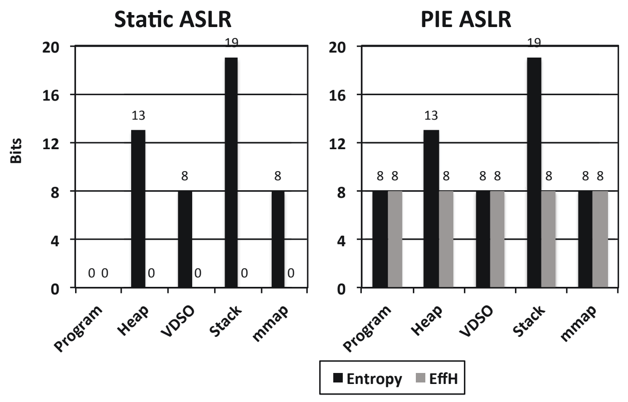

Figure 2: Advanced Adversary threat model. Entropy and EffH results shown for systems protected by static ASLR and PIE ASLR. Note that EffH is 0 for all sections in static ASLR

2. Any non-degenerate execution path in a sufficiently large program uses a set of dynamic connections which is drawn from the same distribution as any other non-degenerate execution path. Thus we do not require an exhaustive search traversing every potential execution in order to ascertain the dynamic connections formed by a program. Rather we traverse only a single execution path in each program.

These assumptions are reasonable because in a given computing environment memory sections are connected in standard ways. For example, on many systems pointers to executable code and readable data are pushed onto the stack while function addresses are written to the Global Object Table. Reasonably long execution paths utilize a variety of library functions, executable instructions, and program data that should be representative of the computing system more generally. Our empirical testing corroborates these assumptions since we find no variation in EffH measurements between measured programs under static and PIE ASLR protection (see Section 7).

We also assume that the permissions of sections in memory do not dynamically change during the runtime of a process.

7 Evaluation Results

Empirical tests of static ASLR on 32-bit Linux machines demonstrate that the stack is randomized with 19 bits of entropy, heap with 13 bits, `mmap` with 8 bits, and VDSO with 8 bits. Specifically, the stack's entropy incorporates kernel-level ASLR applied at the page level for security purposes, and a secondary minor randomization applied in an attempt to avoid L1 cache evictions in certain circumstances. This additional process randomizes the most

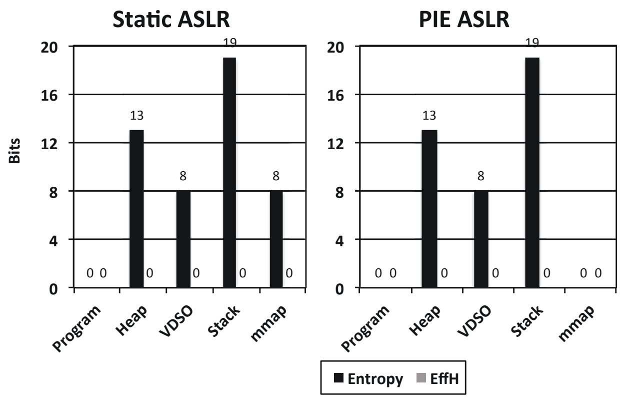

Figure 3: Memory Disclosure Adversary threat model. Entropy and EffH results shown for systems protected by static ASLR and PIE ASLR. Note that EffH is 0 for all sections

significant 8 bits of the non-randomized 12 bits remaining from page alignment (all but the 4 least significant bits). Running a program with the ADDR_NO_RANDOMIZE flag disables both randomization processes and they cannot be otherwise individually disabled. Thus we do not distinguish between these two randomizations processes since an adversary will encounter both identically in the stack.

Figure 2 shows entropy and EffH for all sections in memory in both static and PIE ASLR under the Advanced Adversary threat model. All of the programs we measured provided the same results for static and PIE scenarios respectively. These similarity of results is not surprising since in both static and PIE ASLR the program image has connections to every other section in memory. In static ASLR the program image's zero bits of entropy implies that all other sections have zero bits of EffH. In contrast, in PIE ASLR all sections are reduced to a maximum of 8 bits of EffH.

To simulate the effects of a Memory Disclosure Adversary, we consider the case where a pointer to the program image was pushed onto the stack and then maliciously forced to be printed out. The adversary is expected to know the location of the program image with certainty. While the EffH of static ASLR remains constant since the location of the program image was never randomized this disclosure allows the adversary to reduce the EffH of PIE ASLR as shown in Figure 3. In PIE, the program image is connected to the `mmap`, so knowledge of the program image's base address implies knowledge of the `mmap`'s location. Thus, the `mmap` section has zero bits of entropy since even without any connection from the program image to the `mmap` section, an adversary can know its location with certainty. The remaining sections, which have connections originating from the program image,

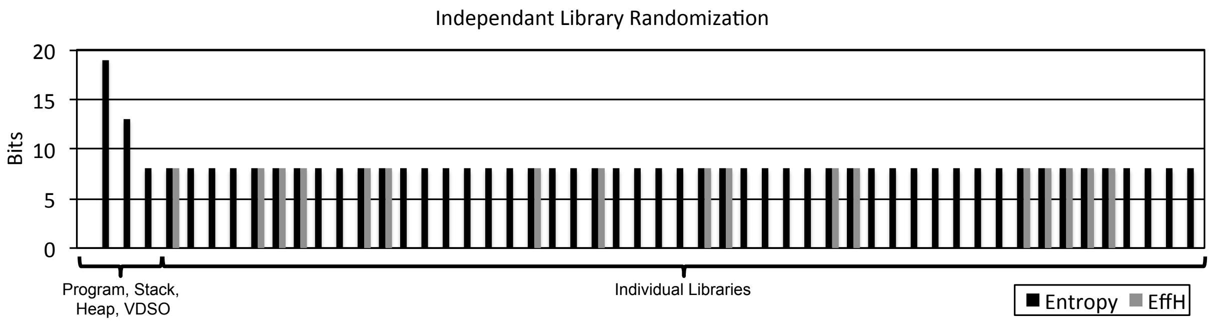

Figure 4: Entropy and EffH measurements of memory sections in systems protected by independent library randomization. Each library section is plotted towards the right side of the x-axis while the program image, stack, heap, and VDSO sections are plotted towards the left. Results shown for the Memory Disclosure Adversary threat model

have zero bits of EffH.

Fine grain randomization attempts to increase the granularity of memory randomization. In addition to increasing the entropy of memory, these techniques attempt to reduce the utility of an arbitrary memory disclosure. They assume that as more sections are independently randomized, a memory disclosure will provide less information about the layout of memory and limit the adversary's resources such as the location of ROP gadgets and system calls needed to construct effective exploits [17].

While no commodity fine grain randomization system exists, we simulated the effects of fine grain randomization in the `mmap` section by assuming that every library is independently randomized with 8 bits of entropy. We refer to this randomization as independent library randomization. Using the previously identified connections we calculated the theoretical EffH of independent library randomization in static ASLR under the Memory Disclosure Adversary. Figure 4 shows typical results of this simulation run on the vim text editor program. Note that a number of individual libraries retain 8 bits of EffH though most have zero bits of EffH.

If we consider only the executable libraries from the vim example and measure the ROP gadgets available in those sections (see Figure 5) we see that adversaries still have thousands of ROP gadgets to use in the sections with zero bits of EffH. Note that the section without entropy is the program image. Similarly Figure 6 shows that most of the sections with system calls still have zero bits of EffH. The independent library randomization fails to prevent adversaries from acquiring knowledge of essential attack elements.

The results from our evaluations suggest that for most sections in a program EffH is reduced to match the section with the minimum entropy. Designers of randomization techniques should consider first focusing on raising the minimum entropy across all sections rather than achieving a higher average entropy or maximizing the entropy of critical sections. A critical section with high entropy often proves to be easily reachable due to connections from the ostensibly less important, lower entropy sections.

8 Conclusion

We introduced Effective Entropy, a new metric to quantifiably assess the security provided by memory randomization techniques. This metric combines the mathematical rigor of traditional entropy measurements with the realistic threat models of sophisticated adversaries by considering connections within memory. EffH provides insight into memory randomization technologies. Buttressed by previous studies, our analysis demonstrates fundamental weaknesses in static ASLR, PIE ASLR, and fine-grain independent library randomization schemes. These analyses illustrated how EffH can be used to simulate the effects of a memory randomization scheme before development and expose potential attack vectors.

While EffH demonstrates the difficulty to reach certain areas of memory, it does not measure the utility of those memory sections to an attacker. Additionally, quantifying the marginal increase in adversary workload with respect to EffH, or another metric, remains an open research question. Little empirical data exists regarding adversary workload and it is difficult to construct high fidelity models.

Future work could evaluate more memory randomization schemes with EffH including those which dynamically re-randomize layout during runtime. Additionally more adversarial models such as those involving just-in-time compilation could be considered in order to expand current understanding of the effects of such techniques.

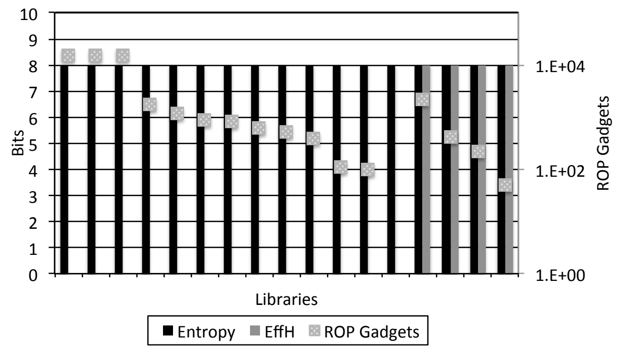

Figure 5: The number of ROP gadgets, bits of entropy, and bits of EffH for each executable library in a system protected by independent library randomization. Results shown for the Memory Disclosure Adversary threat model

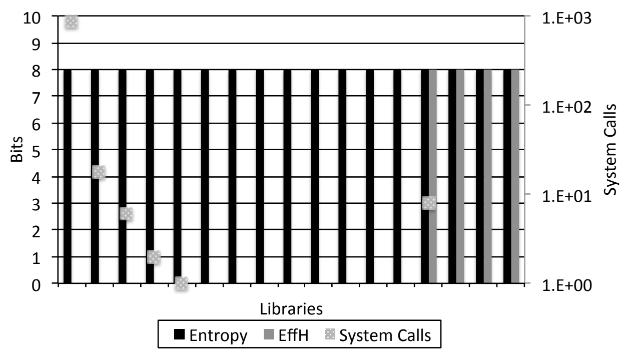

Figure 6: The number of system calls, bits of entropy, and bits of EffH for each executable library in a system protected by independent library randomization. Results shown for the Memory Disclosure Adversary threat model

Acknowledgments

The authors would like to thank David Bigelow and Hamed Okhravi for their work on empirical measures of entropy. This work is sponsored by Assistant Secretary of Defense for Research & Engineering under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

References

[1] Bhatkar, S., and Sekar, R. Data space randomization. DIMVA '08, Springer-Verlag, pp. 1–22.

[2] Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. Jump-oriented programming: a new class of code-reuse attack. ASIACCS '11, ACM, pp. 30–40.

[3] Boehm, H.-J. Space efficient conservative garbage collection. PLDI '93, ACM, pp. 197–206.

[4] Canonical Ltd. Ubuntu security features.

[5] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. Return-oriented programming without returns. CCS '10, ACM, pp. 559–572.

[6] Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. vol. 81 of *USENIX Security '98*, pp. 346–355.

[7] de Raadt, T. The OpenBSD 5.3 release.

[8] Ding, Y., Wei, T., Wang, T., Liang, Z., and Zou, W. Heap taichi: exploiting memory allocation granularity in heap-spraying attacks. ACSAC '10, ACM, pp. 327–336.

[9] Etoh, H., and Yoda, K. Propolice: Improved stacksmashing attack detection. *IPSJ SIG Notes 75* (2001), 181–188.

[10] Giuffrida, C., Kuijsten, A., and Tanenbaum, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security '12*.

[11] Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., and Davidson, J. Ilr: Where'd my gadgets go? In *IEEE S&P '12*, pp. 571–585.

[12] Jackson, T., Homescu, A., Crane, S., Larsen, P., Brunthaler, S., and Franz, M. Diversifying the software stack using randomized nop insertion. In *Moving Target Defense*. 2013, pp. 151–173.

[13] Kc, G. S., Keromytis, A. D., and Prevelakis, V. Countering code-injection attacks with instruction-set randomization. CCS '03, ACM, pp. 272–280.

[14] Kil, C., Jim, J., Bookholt, C., Xu, J., and Ning, P. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. ACSAC '06, pp. 339–348.

[15] Pappas, V., Polychronakis, M., and Keromytis, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE S&P '12*, pp. 601–615.

[16] Schwartz, E. J., Avgerinos, T., and Brumley, D. Q: exploit hardening made easy. USENIX Security '11, pp. 25–25.

[17] Serna, F. J. The info leak era on software exploitation. In *Black Hat USA* (2012).

[18] Shacham, H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). CCS '07, ACM, pp. 552–561.

[19] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. On the effectiveness of address-space randomization. CCS '04, ACM, pp. 298–307.

[20] Snow, K. Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., and Sadeghi, A.-R. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. 574–588.

[21] Sotirov, A., and Dowd, M. Bypassing browser memory protections in windows vista. *Blackhat USA* (2008).

[22] Szekeres, L., Payer, M., Wei, T., and Song, D. Sok: Eternal war in memory. In *IEEE S&P '13*, pp. 48–62.

[23] Wang, T., Wei, T., Lin, Z., and Zou, W. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS '09*.

[24] Wartell, R., Mohan, V., Hamlen, K. W., and Lin, Z. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. CCS '12, ACM, pp. 157–168.

[25] Wojtczuk, R. The advanced return-into-lib (c) exploits: Pax case study. *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e* (2001).