Measuring ASLR Implementations on Modern Operating Systems

David Herrera Aristizabal
Researcher
Universidad de Antioquia
Medellín, Colombia
herreradavid@gmail.com

David Mora Rodriguez
Computer Science Professor
Universidad de Antioquia
Medellín, Colombia
david.mightyd@gmail.com

Ricardo Yepes Guevara
Computer Science Professor
Universidad de Antioquia
Medellín, Colombia
ryepesg@gmail.com

Abstract—**Address Space Layout Randomization (ASLR) is a security technique that uses randomness to conceal regions of virtual memory address space of processes, to increase the effort required to develop reliable exploits. It was introduced in the first decade of the third millennium as a mechanism to stop a dangerous attack vector, known as memory corruption. This paper presents a methodology used to evaluate the effectiveness of this technique, as well as the results of the evaluation of two commonly used operating systems: Microsoft's Windows 7 and Canonical's Ubuntu Linux 10.10, both in the 32-bits and 64-bits versions.**

Keywords— ASLR; Memory corruption; Windows protections; Linux protections

I. Introduction

Memory corruption vulnerabilities were first discussed privately during the 1970s and became a public attack vector in 1988. Haroon Meer presented the history of this attack vector along with some other interesting details in [1]. The main goal of attackers abusing this kind of vulnerability is to manipulate the control data which resides in the virtual memory address space of processes. By corrupting certain data, such as the stored return addresses resulting from the use of the assembly call instruction, stored pointers towards procedures or functions, and others, it is possible to execute arbitrary code and to take control over the victim's operating system. These attacks are dangerous because the arbitrary code executed can be injected; but to do so, the address of the code must be known and used to corrupt the data. For example, if an attacker is able to manipulate a function pointer, it can overwrite its data with the address of the arbitrary code. When the program uses this pointer, it will be calling a user-controlled buffer of instructions.

ASLR is a technique of security through obscurity introduced in the first decade of the third millennium as a mechanism to inhibit memory corruption attacks. Its objective is to obfuscate the virtual memory address space of processes, and it is considered a functional technique because an exploit developer needs to hardcode memory addressees required when memory corruption vulnerabilities are being exploited.

The effectiveness of ASLR has been discussed and evaluated by other researchers as in [2] and [3] and applied to other operating systems. However, the approach of [2] and [3] was different and there is no easy and impartial way to compare the implementation of different systems. Moreover, the criteria used to determine the randomness were not as objective as expected. The method described in this paper, used to determine randomness, uses a more generic and complete set of criteria.

This paper is organized as follows: Section II covers the methodology used to evaluate the effectiveness of ASLR. The evaluation of ASLR in Microsoft Windows 7 appears in Section III. Section IV presents the evaluation of the ASLR implementation of Ubuntu Linux 10.10. Finally, Section V discusses the effectiveness of ASLR on both systems and gives some special considerations.

II. Methodology

Operating systems experience certain limitations on the layout of address space in processes. To mention a few, there is the need for aligning data to page boundaries, the memory splits for user and kernel space, and other control data like Windows' Process Environment Block (PEB) (Microsoft and Linux´s Global Offset Table (GOT). As a result, the memory address space randomized by ASLR is not completely variable and just a portion of the bits of each address is randomized. During this research, different tests and analysis were made on two different ASLR implementations in 32 and 64 bits versions. The purpose was to evaluate common aspects of ASLR in two operating systems and then make a comparison of them.

The number of addresses extracted was at least as many samples as the total number of possible addresses. Then, the number of randomized bits in each address was calculated using the same approach used by the PAX-TEST program [5]. While this measure can give some ideas of the entropy of implementations, by no means, this value can report information about the quality of the randomness produced by the PRNG on ASLR (this is not a correct entropy measure). As a result, more measures were needed. The quality of the PRNG is very important for an ASLR implementation as it will be shown later.

The second step in the methodology was to make a graphical representation of the ASLR behavior, represented by a frequency distribution for key data areas: heap, stack, code and shared libraries. In order to capture the addresses, it is important to consider some special situations: The number of randomized bits affects the number of addresses needed to make a representative frequency distribution graphic. Additionally, some implementations of ASLR require that the operating system reboots in order to produce a new randomization. This happens in Windows 7 as a consequence of the dynamic libraries mechanism and the way they are shared among processes. More information about this is presented in [6].

Finally, in order to perform a deeper evaluation of the quality of randomness in the implementations of ASLR, a statistical test suite was required. Measuring randomness supposes a complicated challenge since there are many different ways a sequence can behave suggesting randomness but some bias towards specific values could remain. The statistical test suite used for this work was presented in NIST 800-22 [4]. There are other statistical test suites for randomness such as Dieharder [7], but NIST 800-22 was selected because it is composed by 15 tests including the best tests from other test suites (including Dieharder) and was written by a trustworthy agency with a renowned background.

The entire test suite presented in NIST 800-22 was used in this methodology. The first test, called Monobit, is the most basic one, and it is mandatory that a sequence under observation passes this test, which means it accepts the null hypothesis, (the sequence is random) in order to run the subsequent tests. Therefore, if a sequence fails this test, the sequence is not considered random and no further evaluation is required. Every test defined in NIST 800-22 returns a P-value, which is a measure of the strength of the evidence that a sequence is random. If the P-value is higher than a significance level, which is the measure of the probability of mistakenly rejecting the null hypothesis, the sequence is accepted as random under that test. During this research the significance level used was 0.01 since [4] suggests that the significance level should be between 0.01 and 0.001. There are some difficulties in using an extremely powerful PRNG for ASLR. Due to the fact that processes are created at different rates on modern computers, it is important that the seed of entropy that generates randomness in the rest of applications (such as cryptography tools) is not drained; therefore, ASLR implementations tend to trade off the quality of the seed to preserve entropy, and for this reason the lowest significance level recommended in [4] was selected.

Each P-value allows the evaluation of different aspects of the sequence and gives ideas of how to circumvent the defense technique. Additionally, when any of the P-values was lower than the significance level, the area was not considered randomly allocated.

III. Evaluation of Windows 7 sp0

In this section, the evaluation for Windows 7 Ultimate Sp0 in 32 and 64 bits architectures is presented. The operating systems were running in a virtual machine (VirtualBox 3,2,10 r66523).

As heap management is a user level responsibility, an operating system can include different mechanisms (in form of user level libraries) to perform this management. In order to evaluate common aspects of different operating systems, the heap allocations were performed using the standard C call to malloc that resides in msvcmrt. The first step in this methodology is to acquire the quantity of bits randomized for each virtual memory region.

TABLE I. Number of bits randomized for each operating system that was evaluated.

Region	Linux 32	Linux 64	Windows 32	Windows 64
Main (code)	0	0	9	8
Heap	14	14	7	8
Stack	20	29	17	14
Shared Lib	12	29	9	9

Table I presents an intriguing fact: the number of bits randomized in Microsoft implementations is very low, even for its 64 bits version. As a matter of fact, there were more randomized bits encountered in the 32 bits version. This is something unexpected, and in order to detect errors in the measurements, the procedure was repeated in a different machine obtaining the same results. Other authors had detected a flaw in ASLR when low quantities of bits are randomized [2-3], [8]. This makes brute force and guessing attacks trivial.

A. *Frequency distributions of Windows 7 sp0*

Figures presented in the appendix, have addresses on the X axis and frequency of apparition on the Y axis. The frequency distributions that exhibit more randomness are the shared libraries for both versions of the operating system (figure 1 and 2) and the code region on the 64 bits version (presented in figure 4). However, there is a design flaw in the libraries randomization used in Microsoft's operating system. The libraries are randomized only once for each reboot giving the attacker an advantage to brute force or guess addresses in this region. It even makes return oriented programming feasible, which increases the chance to bypass the no execute (NX) bit.

In the appendix, figures 5 and 6 present the frequency distributions for the stack in Windows 7; it should be noted that it appears as if the distributions were discrete leaving the stack exposed to brute force attacks. Figures 7 and 8 show the heap for Microsoft's ASLR implementation and indicate that the distribution is far from being uniform.

The frequency distribution for the ASLR randomized areas raises many questions about Microsoft's implementation of ASLR. The results are comparable with Whitehouse's work in [3] and they definitely show a tendency to more probable values. This means that known weaknesses in the ASLR implementation of Windows Vista were not corrected in the development of Windows 7.

The distribution of the heap for 32 and 64 bits architectures was very similar and presented some bias. It is interesting to note that the shape of the distribution of the heap was similar to the one obtained by Whitehouse in his analysis of Windows Vista [3]. These results suggest that an attacker can select some addresses and expect a greater probability of success.

B. *Evaluation of Windows 7 sp0 using NIST 800-22 Statistical test suite.*

In table II, the P-values for the regions that passed the Monobit test are listed. Notice that for Windows, only the stack distribution passed the first test (Monobit), but obtained 0´s in 8 tests making this memory area nonrandom. It is important to highlight that none of the areas in the 64 bits version passed the Monobit test.

IV. Evaluation of Ubuntu-Linux

To evaluate the Linux kernel, 32 and 64 bits versions of Ubuntu GNU/Linux 10.10 (using Kernel 2.6.35-23-generic SMP) were used. The ASLR implementation tested was the one integrated into the kernel and based on the Arjan Van de Ven's code [9].

The number of variable bits in each region was presented in table I in section III. It is important to note that the code region is not randomized in Ubuntu-Linux. Even though this feature is available as a compiler option, this is not by default and a lot of binaries are built without code randomization; therefore, it was decided not to enable this feature for this test. On the other hand, the stack and the shared libraries presented an important increase of randomized bits in the 64 bits when compared with the 32 bits version.

A. *Frequency distributions of Ubuntu-Linux*

The Appendix presents the Ubuntu GNU/Linux´s frequency distributions. Ubuntu Linux showed a better randomization mechanism than its counterpart. However, the response of ASLR for the shared libraries of Linux 32 bit had a notorious lack of randomness. It was biased towards 0x157B20 (Figure 9 in the Appendix). This fact is alarming, since it makes return oriented programming attacks feasible. The other regions showed an expected graphical behavior for random distribution.

B. *Evaluation of Ubuntu Linux using NIST 800-22 Statistical test suite.*

Table II shows the P-values for Ubuntu-Linux. Only the regions that successfully passed the Monobit test are presented.

For the 32 bits version, the heap and the shared libraries did not pass the evaluation. In fact, they did not even pass Monobit test. On the other hand the stack and the shared libraries got all their p-values over the significance level which means they are randomly allocated. The heap did not pass the Monobit test in either of the architectures evaluated.

TABLE II. P-VALUES FOR THE STACK ON LINUX 32 BITS, STACK AND SHARED LIBRARIES ON LINUX 64 BITS.

Test	Stack Win 64 b	Stack Linux 64 b	Sh lib Linux 64 b	Stack Linux 32 b
Frequency or Monobit	0.225	0.644	0.941	0.501
Frequency Test within a Block	1	0.087	0.749	0.274
Runs	0	0.174	0.824	0.715
Longest Run of Ones in a Block	0	0.159	0.264	0.716
Binary matrix rank	0	0.192	0.177	0.139
Discrete Fourier Transform (Spectral)	0	0.057	0.339	0.538
Non-overlapping Template	0.057	0.510	0.438	0.464
Overlapping Template	0	0.678	0.305	0.397
Maurer's "Universal Statistical"	0	0.232	0.313	0.286
Linear Complexity	0.237	0.172	0.684	0.426
Serial	0	0.633	0.145	0.023
Approximate Entropy	0	0.744	0.778	0.339
Cumulative Sums or Cusum	0.007	0.725	0.914	0.656
Random Excursions	0.443	0.497	0.467	0
Random Excursions Variant.	0.560	0.399	0.438	0

V. Final Thoughts and Conclusions

The stack presents the higher number of variable bits in all the tests (Table I). Therefore, it is the most protected area in terms of randomness and non-execution. However, it is worth mentioning that given the no-execute protection on modern operating systems the stack is not as abused as it was before. Although new attacks try to avoid the no-execute restriction using variations of return to library approaches, such as return oriented programming; ASLR is a key factor to defend against this technique.

Definitely, there are some flaws in the PRNG used to obfuscate the memory regions of processes. The statistical analysis presented hereby contains information that might allow an attacker to build attacks with higher probability of success. In fact, it is more important to evaluate the meaning of each test result on the NIST document than to make decisions based on the P-values. These decisions can help an attacker to increase the propagation of malware that uses memory corruption as the method of infection. This is even worse when the attacker is targeting a large number of victims, for example while creating a botnet, since the attacker will benefit from partial success of his attack.

Apart from the randomness issues discussed, there are some weaknesses in the ASLR implementations evaluated. First, Microsoft's ASLR depends on the compile option /dynamicbase which is not included in all the compilers available. Second, during the evaluation, there was evidence that compilers such as GCC do not use an equivalent for /dynamicbase; and therefore, applications built with this tool are not protected with ASLR. Currently, there is a lot of software being distributed with no ASLR protections. Finally, to make things worse, not only the final binary code has to opt-in the randomization, but also third party libraries need to enable the feature as well. Developers have two options while constructing software for Microsoft's platform. One is to use the latest Microsoft´s compilers, which fully support out-of-the-box ASLR. Another is to use a compiler that includes an equivalent flag to enable this security feature. The other is to make a direct modification of the PE that can enable ASLR for binaries built with a compiler that does not support it.

GNU/Linux has its drawbacks as well. First, in order to preserve compatibility, a global randomization interface exists under /proc/sys/kernel/randomize_va_space. This interface can be configured as 0- No randomization, 1 - Randomize everything but heaps, 2 - Full randomization. The problem is that not all distributions enable level 2 by default, and as a result the heap is exposed to attacks. Second, in order to enable randomization on the code region, the binary code has to be built as an independent position code which is a compiler option.

The main problem with security features as compile options is that the developer should be knowledgeable enough in security to activate them. There will always be more newcomers to programming than experienced developers. Therefore, it is important to integrate these requisites in programming courses and books, and to train developers in secure application development.

APENDIX

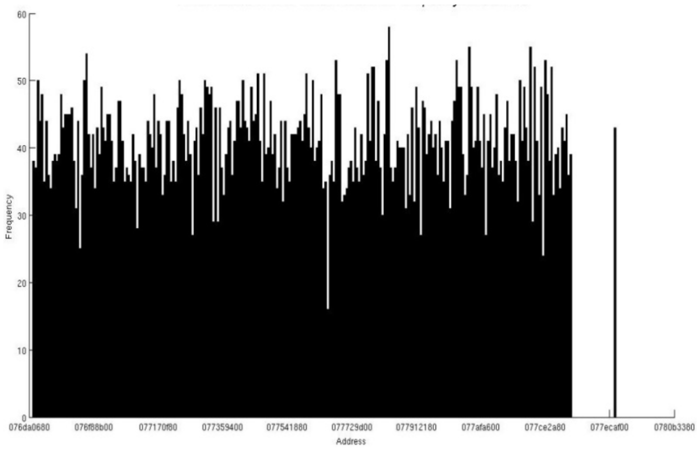

Fig. 1. Libraries area on Windows 7, 32 bits SP0

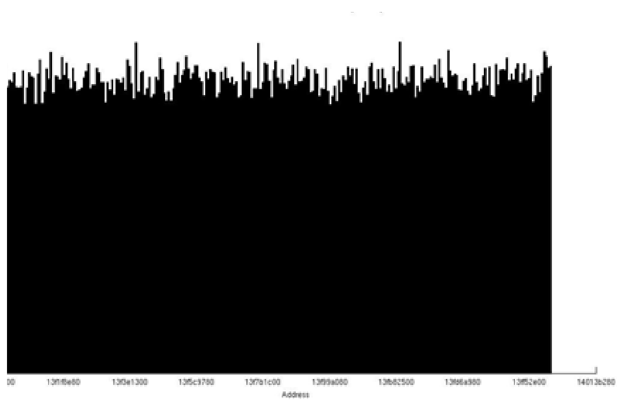

Fig. 2. Libraries area on Windows 7, 64 bits SP0

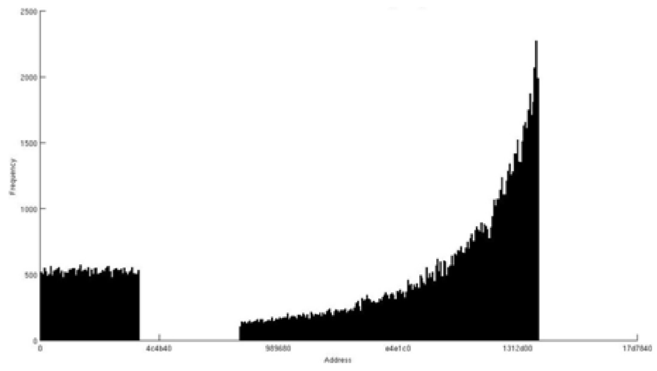

Fig. 3. Code area on Windows 7, 32 bits SP0

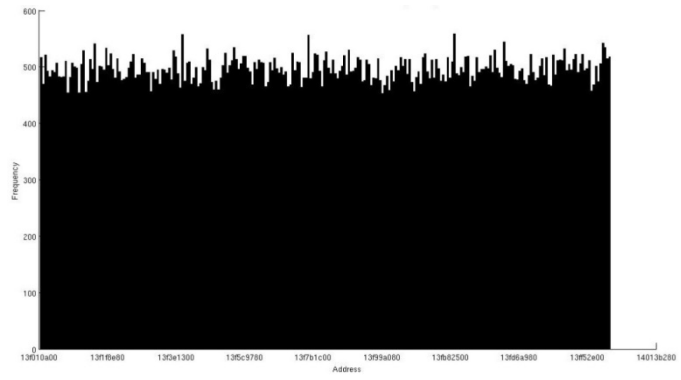

Fig. 4. Code area on Windows 7, 64 bits SP0

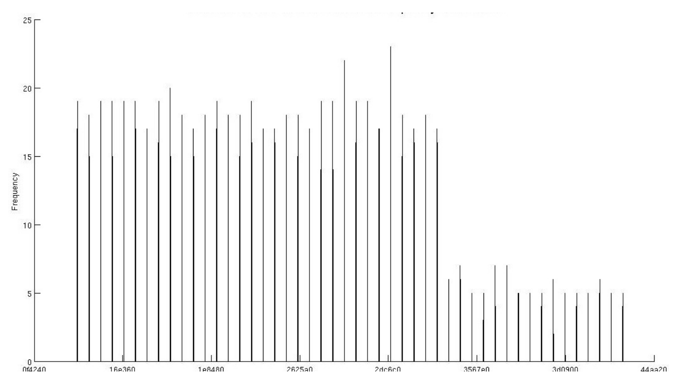

Fig. 5. Stack area on Windows 7, 32 bits SP0

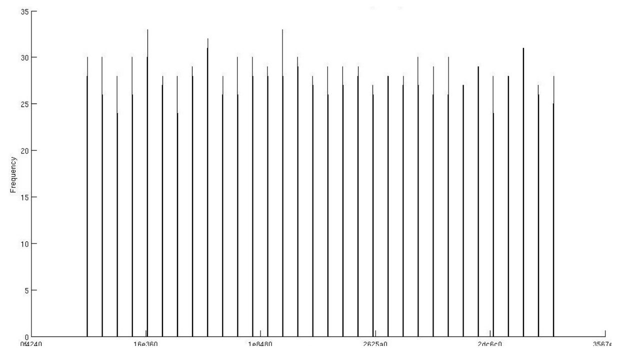

Fig. 6. Stack area on Windows 7, 64 bits SP0

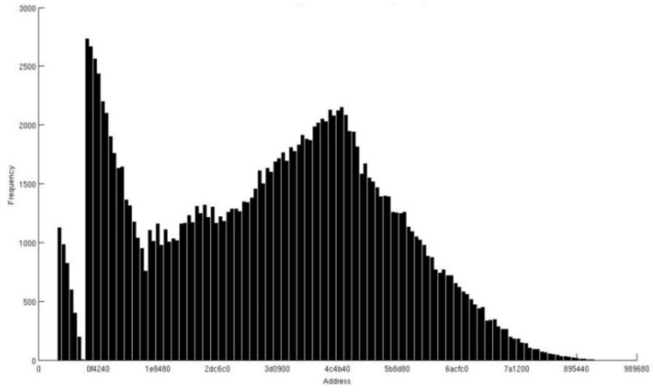

Fig. 7. Heap are on Windows 7 SP0 32 bits

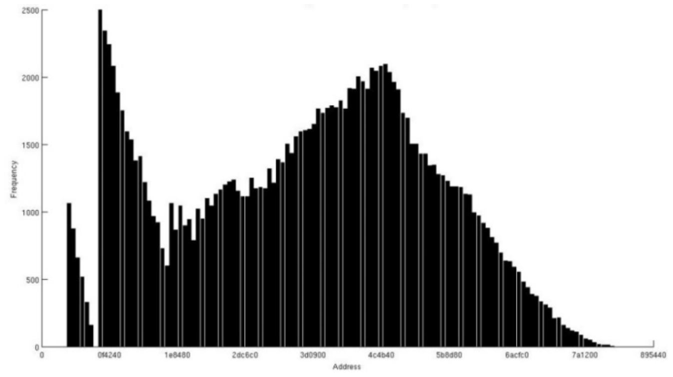

Fig. 8. Heap area on Windows 7 SP0 64 bits

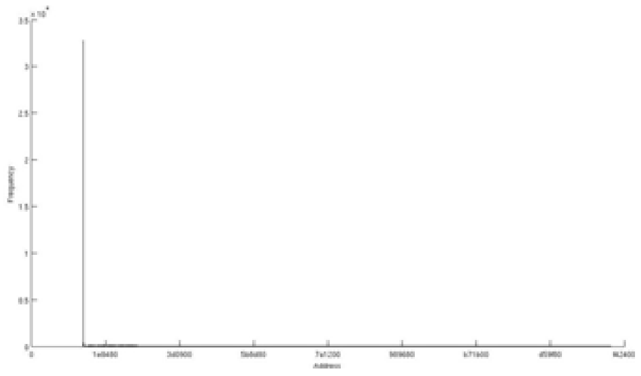

Fig. 9. Shared Libraries area on Ubuntu Linux 32 bits

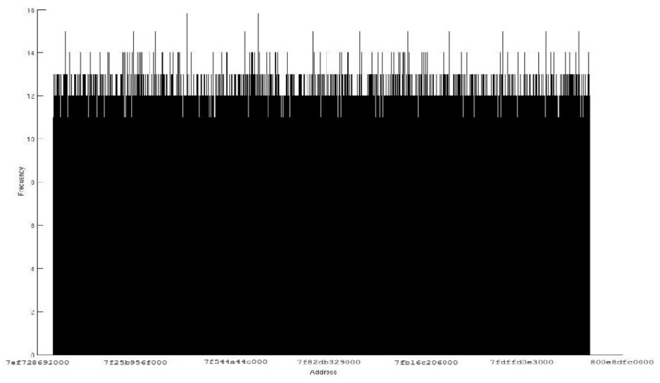

Fig. 10. Shared Libraries area on Ubuntu Linux 64 bits

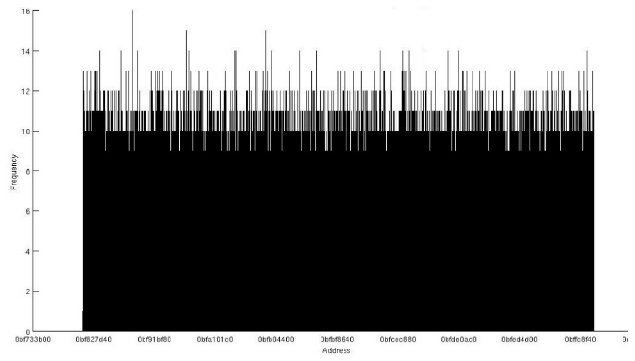

Fig. 11. Stack area on Ubuntu-Linux 32 bits

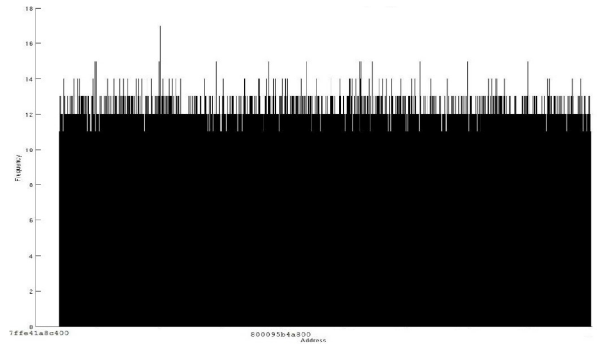

Fig. 12. Stack area on Ubuntu-Linux 64 bits

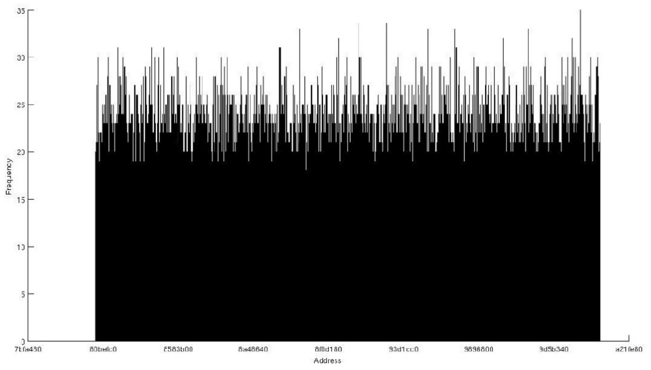

Fig. 13. Heap area on Ubuntu-Linux 32 bits

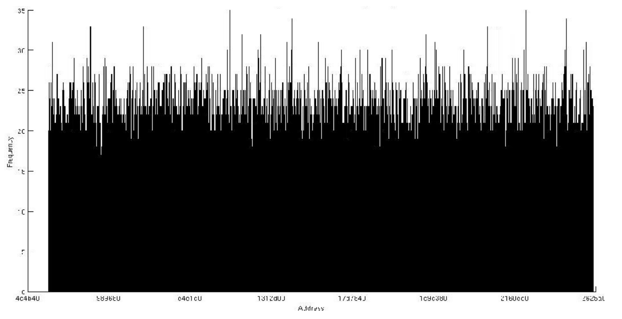

Fig. 14. Heap area on Ubuntu-Linux 64 bits

REFERENCES

[1] Haroon Meer, "Memory Corruption Attacks, The (almost) Complete History", Available on: http://thinkst.com/stuff/bh10/BlackHat-USA-2010-Meer-History-of-Memory-Corruption-Attacks-wp.pdf, (2010)

[2] H. Shacham, M. Page, B. Pfaff, E. 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 (CCS 2004), 2004.

[3] Ollie Whitehouse, An Analysis of Address Space Layout Randomization on Windows Vista, Available at: http://www.symantec.com/avcenter/reference/Address_Space_Layout_Randomization.pdf , Symantec,(2007)

[4] Andrew Rukhin et al. NIST 800-22 Rev 1a A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications, National Institute of Standards and Technology. Technology Administration U.S Department of Commerce, (2010)

[5] Brad Spengler. Pax: The guaranteed end of arbitrary code execution, G-Con2: Mexico City, Mexico, (2003)

[6] Lixin Li, James E. Just, R. Sekar: Address-Space Randomization for Windows Systems, 22nd Annual Computer Security Applications Conference, ACSAC '06, p. 329 -338 Defense Advanced Research Project Agency, USA (2006)

[7] Robert G. Brown: Dieharder, Duke University Physics Department Durham, NC 27708-0305 available at http://www.phy.duke.edu/~rgb/General/dieharder.php

[8] Ali Rahbar: An analysis of Microsoft Windows Vista's ASLR, sysdream (2006) available at :

http://www.sysdream.com/sites/default/files/Analysis-of-Microsoft-Windows-Vista%27s-ASLR.pdf

[9] Arjan Van de Ven: Patch virtual address space randomization, available at: http://lwn.net/Articles/120966/, http://lwn.net/Articles/120967/ http://lwn.net/Articles/120968/, http://lwn.net/Articles/120969/ http://lwn.net/Articles/120970