Universitat Politècnica de València

CyberSecurity research group
http://cybersecurity.upv.es

Exploiting Linux and PaX ASLR's weaknesses on 32- and 64-bit systems

Author:
Dr. Hector Marco-Gisbert

Author:
Dr. Ismael Ripoll-Ripoll

Black Hat Asia
March 29 - April 1, 2016, Singapore

Contents

Abstract

Address Space Layout Randomization is a very effective mitigation technique. The first implementation was done by the PaX team in 2001, and since then it has been the most advanced and secure. We have analyzed the PaX an Linux implementations, and found several weaknesses.

We have carried out a deep review and analysis of all constraints that determine ASLR operation. Based on these results we have designed and implemented a novel ASLR called ASLR-NG, which maximized the entropy (security) and does not introduce fragmentation (compatibility). ASLR-NG is specially suitable for 32-bit systems because of their intrinsic reduced VMA size.

We have developed ASLRA, a tool to analyze the quality of the ASLR. This tool shows that ASLR-NG outperforms PaX ASLR in all aspects.

1 Introduction

Despite the large amount of research effort trying to create programs free of errors, bugs exist. We all are humans, and we all make mistakes, and so error mitigation is necessary. The most widely used memory error mitigation techniques are: "Non-eXecutable or Data Execution Prevention (NX)", "Stack Smashing Protector (SSP)" and "Address Space Layout Randomization (ASLR)" (another mechanism also used but not widely known is "pointer mangling"). All those mitigation techniques were presented and extensively developed during the first decade of this century. Today, they are considered as well studied, well understood and even "boring" stuff by the security community.

There is a huge gap between an abstract concept and its actual implementation. Flying is easy on the paper, just a pair of big wings and a propeller with sufficient trust; but it is not that easy on the real world. Any change and improvement on the available technology to build planes may introduce radical changes in the way planes are designed and implemented.

The core ideas of all those techniques where stated more than 15 years ago, and although they are perfectly valid today, it is mandatory to revisit and reevaluate those techniques as technology evolves. The world has changed a lot since 2000: wider word size, faster processing, smarter and more motivated hackers, etc. Just to mention a recent contribution to the SSP, we presented a small modification of the SSP which eliminates brute force attacks against stack guard on forking servers (Apache, Zygote (Android), etc.).

This white paper presents a critical review of the ASLR design and implementation on Linux and PaX, points out several weaknesses and proposes a practical solution.

2 Basic ideas of the ASLR

ASLR is one of the most effective memory error mitigation techniques. Once an attacker is able to redirect the control flow of the target process, the next step is to jump to the malicious code (injected, ROP, reuse an exiting function, etc.). In order to do so, they need to know the exact location (absolute address) of the malicious code[1]. Here is where the ASLR plays a role. The memory positions where the different segments of the process are allocated in the VMA[2] are not fixed but random, and so the attackers don't know the address to jump to. **ASLR relies on keeping secret the virtual memory layout.** For example, the recent vulnerability in the `getaddrinfo()`, presented by Fermin J. Serna [1], requires to bypass the ASLR in order to exploit it.

Just for completeness: the randomization must be applied to all the objects of a process, specially to objects marked as executable (libraries, executable, JIT code, ...); otherwise, an attacker can use the non-randomized object to bypass the ASLR. In order to randomize the executable image, it must be PIE[3] compiled. In the Fedora distribution, most of the executables are PIE, while in Ubuntu, they are not. In our modest opinion, all 64-bit applications shall be PIE compiled, there is no reason why not to do it. A discussion of PIE vs. non-PIE is beyond the scope of this paper. In what follows, we will assume that all the applications are PIE compiled, that is, the executable is also randomized.

Due to the multiple constraints, each object may be differently randomized. Since an attacker may try to abuse the "least" randomized object we shall focus on those weakly randomized objects to determine the ASLR security of a system.

[1] Not always it is necessary to know the absolute address. For example, relative addressing or spraying may alleviate this problem.

[2] VMA: Virtual Memory Area or space.

[3] PIE: Position Independent Executable.

The basic idea of the ASLR consists in placing the objects randomly in the virtual memory space. We are not talking about OOP objects, but about process memory regions, areas, zones, blocks or chunks. We will use the term **memory object** (or object for short) to refer to "*a continuous range of virtual memory with the same attributes*": the stack, a mmapped file, the heap, the code of a library, the executable image, etc.

Obviously, the more "random" the memory objects are placed the more difficult will be to the attacker to guess where they are. Ideally, each object shall be randomly placed at any address of the whole VMA. Unfortunately, there are several constraints that limit the effective randomness of the ASLR.

2.1 Fragmentation problems using the virtual memory

One of the most critical constraints is the **fragmentation problem** which is generally defined as: "*the inability to use memory that is free*". Although objects are allocated in virtual addresses, once an object is mapped in a position, the application will use the given address to access to it, and so it is not possible to move it to another addresses unless the application is completely aware of it (typically, it is not the case). As a result, VMA will get "fragmented" if objects are randomly mapped. Eventually, a new object allocation would fail when there is not a free range of virtual memory large enough to hold it.

Note that virtual memory is a very nice mechanism to deal with "physical" memory fragmentation. But nothing prevents from suffering fragmentation on the VMA on itself. It may look confusing, but ASLR may cause very bad fragmentation problems. And this is one of the main restrictions to the ASLR. In 64-bit systems, the huge difference between the VMA size and the size of the allocated objects makes the fragmentation problem not a big issue, but in 32-bit system it is a real problem. In fact, the fragmentation was the most limiting factor on the design of the ASLR in 32-bits.

Historically, the general fragmentation problem has been (and still is) a challenging issue with many misconceptions. For example, the best-fit allocation strategy produces, contrarily to what may be expected, the worst case fragmentation. Any discussion about fragmentation is far beyond the scope of this paper. The reader is referred to the outstanding paper: "*The Memory Fragmentation Problem: Solved?*" by M.S. Johnstone and P.R. Wilson [2].

2.2 Entropy

Entropy is generally defined as the amount of disorder. In information theory, entropy is used as a measure of the amount of information provided by a piece of data, which typically it is probability that such data is received.

Entropy shall be considered as the level of uncertainty that an attacker have about the location of a given object. There are several expressions that try to measure the amount of entropy of something (stream of video, compressed file, values returned by a RND generator, the values where an object can be mapped, etc.). For our purposes, the most interesting entropy estimator is the Shannon entropy, which is calculated by the following expression, where $p(x)$ is the estimated probability of the address x:

> *"You should call it entropy, for two reasons. In the first place your uncertainty function has been used in statistical mechanics under that name, so it already has a name. In the second place, and more important, nobody knows what entropy really is, so in a debate you will always have the advantage."*
>
> Conversation between Claude Shannon and John von Neumann regarding what name to give to the attenuation in phone-line signals.
> Wikipedia: https://en.wikipedia.org/wiki/Entropy.

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_2 p(x)$$

The value of $H(X)$ can be interpreted as the number of bits that are "unknown", which is a good measure of dispersion or "surprise", but it must be interpreted with caution.

Remember that this expression was used in the context of data communications, and so it perfectly models/measures the capacity of a channel to transfer data, but it does not take into account the actions that a malicious attacker can do. In other words, if the distribution of the values is not uniform, then an attacker may develop an strategy that improve its chances to win.

Although it would be very nice to give a single value to measure the "protection" provided by an ASLR on a given system, there are several other elements that must be also checked. We have developed a tool to perform a complete statistical analysis called ASLRA (described in section 6). All the results presented in this paper has been obtained using this tool.

3 Weaknesses in Linux and PaX ASLR

In this section we describe four weakness present in current Linux and PaX ASLRs.

3.1 Low entropy

Before ASLR, the memory layout of the process was fixed and well defined: the STACK at the far top of the VMA, the executable image at the bottom, the HEAP right above the EXEC and the libraries and mmapped objects somewhere between the HEAP and the STACK. Figure 3.1 shows a picture how the memory map of a process is typically drawn in operating system books. Systematically, all objects are displayed as relatively large rectangles, and the gaps between them are relatively small. The reality is just the opposite.

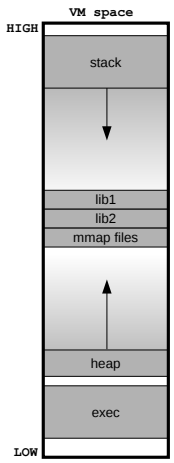
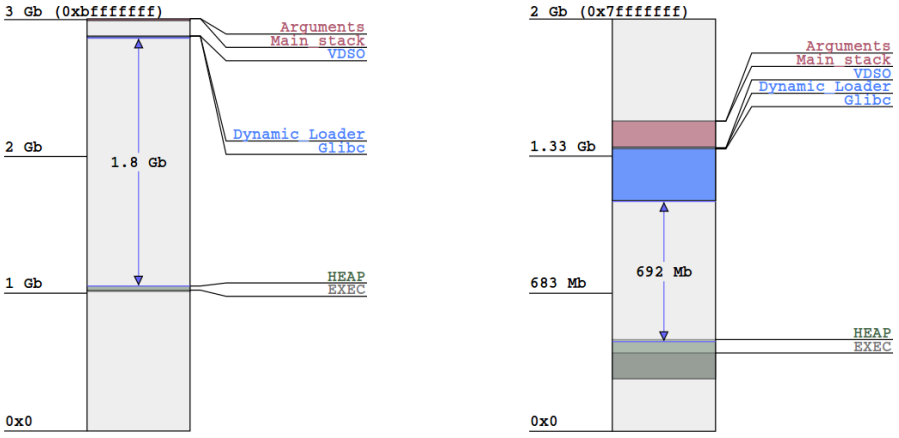


Figure 1: Memory layout on teaching books.

The stack hardly grows more than 8MB; In fact current Linux versions limits the stack to 8MB via ulimit facility. The heap structure is used to store malloc/free allocations but when a single large block of dynamic memory is requested (more than 128KB), a more flexible mechanism to allocate memory is used by the Glibc. We will cover this issue later.

The ASLR has been implemented adding some randomness to the original positions of the objects. In order to maintain the relative position of each object, allow growable objects (stack and heap) and also to avoid fragmentation. As a result, the VMA space used to randomize each object is small with respect to the full VMA. Figure 2 shows the memory layout of Linux and PaX on native 32-bit systems. In this figure, the colored rectangles represent the observed (measured) VMA range where an object has been mapped in 100.000 processes. For example, the blue rectangles correspond to the mmap zones, which are where libraries and mmaps are mapped.



(a) Linux 32 (b) PaX 32

Figure 2: Actual memory (The height of the objects represent their entropy, not their size!)

As can be seen, the VMA space used by PaX for randomizing libraries is far bigger, and so PaX has much more entropy than Linux for these objects. Unfortunately, the largest allocatable object in PaX is only 692MB, which may break some applications. Some multimedia applications use to mmap media files in 1GB chunks, and so it will crash on a 32-bit PaX secured system.

Besides compatibility issues, there are many other factors that affect the layout; for example, PaX implements a smart technique called VMA mirroring which divides the available VMA in two, this technique has several applications (randomize non-PIE applications and implement the NX using segments) which are beyond the scope of this paper.

Figure 3 summarizes the entropy of Linux and PaX in 32- and 64-bits. Some objects have a non integer entropy value, which is caused in most cases by a non-uniform distribution, as it is explained in the next section.

In 32-bits, PaX ASLR is superior to Linux in all objects. Page alignment for Hugepages greatly reduces the bits of the address that can be randomized, the entropy reduction is so important in 32-bit Linux that it does not have randomness in huge pages. It is interesting to see that PaX uses the sub-page randomization in the Arguments[4], HEAP and the Main stack. Linux also implements sub-page, but only in the Main stack object.

Update Trials x sec: 1000

	[Detailed] [Summary] PaX 3.14.21		[Detailed] [Summary] Linux 4.5.0	
Object	**Entropy**	**Time**	**Entropy**	**Time**
Arguments	27.0	1 days	11.0	2 secs
HEAP	23.4	3 hours	13.0	8 secs
Main_stack	23.0	2 hours	19.0	8 mins
Dynamic_Loader	15.7	53 secs	8.0	0 secs
VDSO	15.7	53 secs	8.0	0 secs
Glibc	15.7	53 secs	8.0	0 secs
MAP_SHARED	15.7	53 secs	8.0	0 secs
EXEC	15.0	32 secs	8.0	0 secs
MAP_HUGETLB	5.7	0 secs	0.0	0 secs

(a) 32-bit (x86) system.

Update Trials x sec: 1000

	[Detailed] [Summary] PaX 3.14.21		[Detailed] [Summary] Linux 4.5.0	
Object	**Entropy**	**Time**	**Entropy**	**Time**
Arguments	39.0	17 year	22.0	1 hours
Main_stack	35.0	1 year	30.0	12 days
HEAP	35.0	1 year	28.0	3 days
Dynamic_Loader	28.5	4 days	28.0	3 days
VDSO	28.5	4 days	21.4	46 mins
Glibc	28.5	4 days	28.0	3 days
MAP_SHARED	28.5	4 days	28.0	3 days
EXEC	27.0	1 days	28.0	3 days
MAP_HUGETLB	19.5	12 mins	19.0	8 mins

(b) 64-bit (x86_64) system.

Figure 3: Comparative summary of the observed entropy. Green and red indicate the best and worst randomized object per row.

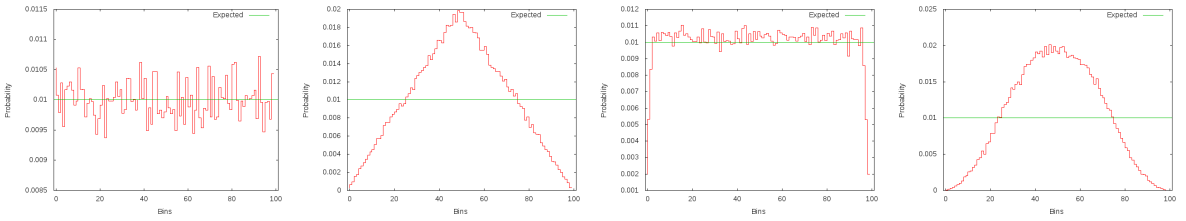
The differences are not that large in 64-bits. PaX has the lowest entropy on one of the most critical objects, the executable (EXEC). This fact could go unnoticed due to the bug in the paxtest tool, see appendix A.2.

It would be easy to add sub-page randomization in the Arguments and the HEAP, and improve the randomization of the vDSO of Linux to have the same security than PaX.

3.2 Non-uniform distribution

When we work with random numbers, unless stated explicitly, we all assume that they follow a uniform distribution, for example, rolling a single not loaded die. But, working with random numbers is always challenging.

The Linux and PaX ASLR implementations are a good example of this situation. The pseudo random number generation used to map objects is not biased (AFAWK) but the final observed entropy of the objects is not as good as expected. We have observed several non-uniform distributions: triangular, trapezoidal, Irwin-Hall (sum of several uniforms) and several irregulars.



(a) Example of a uniform. (b) Pax 32 on libs and mmaps (triangular) (c) Linux 32 on the HEAP (trapezoidal) (d) Pax 64 on libs and mmaps (Irwin-Hal)

Figure 4: Cumulative Distribution Function (CDF) the randomness on selected objects.

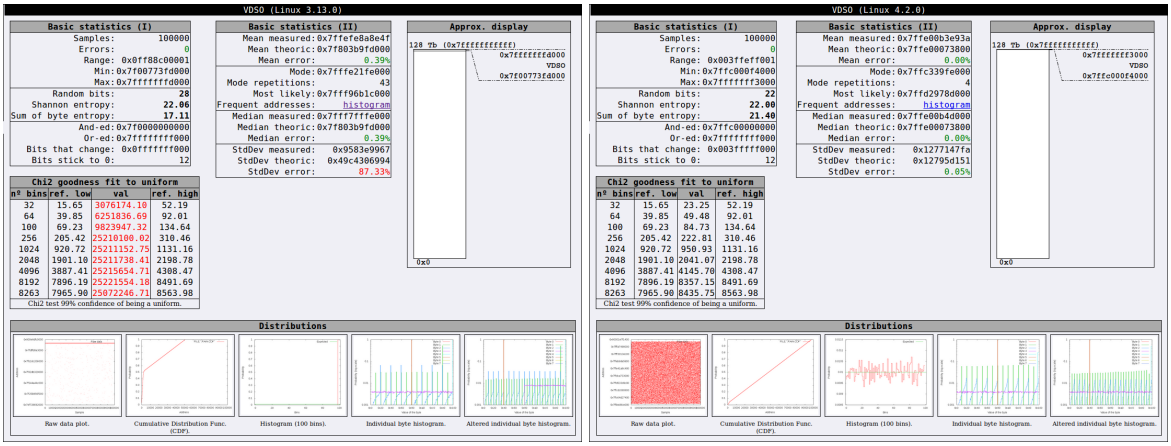
An example of non-uniform is showed in figure 4(b). The libraries and mmaps in 32-bit system in PaX are not uniformly distributed in the range. Actually, they follow a triangular distribution where the values in the center are much more likely. The same weakness is present in PaX 64-bit systems as shown in figure 4(d). This effect is because the final position is calculated as the sum of two (triangular) or three (Irwin-Hall) random values respectively.

Regarding Linux, we observed also non-uniform objects but much less pronounced. As figure 4(c) shows, the HEAP follows a trapezoidal but has much less concentration of the values in the center than

[4]Arguments: the object which contains the argument (ARGV) and environment (ENV) vectors passed to a process.

PaX. Linux heap is the result of the sum of two random numbers of different ranges. The sum of two randoms variables of the same range results in a triangular.

A much worst case was the entropy of the vDSO[5] in Linux. In order to avoid TLB misses, the vDSO was allocated to be in the same PMD that the main stack, which unfortunately resulted in very pour entropy[6].



(a) vDSO affected by CVE-2014-9585 bug.

(b) vDSO after it was corrected.

A detailed analysis of the vDSO of the affected system showed that most of the values are concentrated in a very narrow upper band (see the raw data plot of fig 5(a)). The estimation of the entropy of a very biased and non-continuous function is far from trivial: 28 random bits, 22 using the Shannon expression, and 17.11 when it is measured as the Shannon entropy of individual bytes. But a more accurate value is 11 bits (which is not computed by the tool). The CDF plot shows that a very small set of values have the 50% of probability of occurrence. This example, clearly shows that the quality of the ASLR must not be measured using a single statistical parameter.

Let's suppose that we need to bypass the ASLR on a mobile phone. It is a 32-bits and protected with PaX ASLR, and the vulnerable application needs to know where is located the Glibc (to setup the ROP). In this scenario, the attacker must always try the address that is in the top of the triangle (`0xa7dd3000` or `0xa5053000`). Trying always the same address doubles the probability of bypassing the ASLR, with respect to sequentially trying all the possible values. And obvious, in the case of a brute force against a single application which uses the forking model (as for example android applications, or Apache in forking mode), the attacker shall start by trying the central values and then move to the outside of the range.

Fortunately, PaX libraries (which are triangular on 32-bits and Irwin-Hall on 64 bits) only have one bit less than claimed, which is not too much.

Note that a non-uniform distribution does not automatically makes the ASLR useless. But:

1. The Shannon entropy is not longer a very precise measure of the cost of an attack.
2. An attacker can design a winning strategy to reduce the cost of an attack.

In fact, it is very likely that the entropy of the objects get degraded as the application requests objects. A good distribution shall be as flat as possible and without higher probable values. This way, an attacker can not focus its attack on more likely values.

3.3 Correlation between maps

If knowing the position of an object (an address belonging to some object) gives information (address bits) about where another is in memory, then we said that they are correlated. According to how an attacker can use the correlation, we can define three types: total, partial and useless.

[5] vDSO: is a special page mapped automatically by the kernel in the process VMA, which implements non critical services that can be implemented in user space.

[6] https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-9585.

3.3.1 Total correlation

If knowing the position of an object gives the exact position of another one, then these two objects are said to be "totally correlated". This is the most obvious, known and dangerous correlation and can be found in current ASLR implementations including Linux and PaX.

The simplest example to illustrate this correlation can be found in how current libraries are mapped in Linux and PaX. All libraries are mapped side by side, and so de-randomizing one library gives the positions of all the others (for a given application). Another very well known example in Linux, is when the `randomize_va_space` is set to 1. In this case, the executable and the HEAP are mapped side by side, so, if the attackers leak an address belonging to the heap the they can directly de-randomize the executable and the other way around.

3.3.2 Positive correlation

The second type of correlation is when knowing the position of an object gives us enough information to guess another one with less effort than guessing the second one directly. We name this "positive correlation", because a leak of an address belonging to an object reveals some information (some bits) about where the other object is mapped in memory. Unlike total correlation, we have not enough information to know where other object is in memory but reduces the cost to know it.

For example in 64-bit Linux systems, when the `randomize_va_space` is set to 2, the executable has 28 bits of randomness and the HEAP is located at an address which is the base of the executable plus a random offset of 13 bits. So, the executable and the HEAP have positive correlation: a leak of an address belonging to the HEAP reduces the effort need to de-randomize the executable from 28 to 13 bits.

PaX also suffers from positive correlation, an example of it is between Hugepages and libraries, which reduces the cost of finding the libraries by 9 bits, once the attacker knows where a Hugepage is placed. The details of the affected systems, objects etc., are out of scope of this paper.

3.3.3 Useless correlation

The useless correlation is when two objects, although being mathematically correlated, knowing the position of one of them gives almost no information about the position of the other.

An example of this correlation is present in 32-bit Linux systems between the executable and the HEAP. Contrarily to 64-bit systems, the correlation between the executable and the HEAP is not useful for the attackers. In 32-bit Linux systems, the executable has 8 bits and the heap 13 more with respect to the executable. Therefore, if the attackers obtain an address belonging to the HEAP and they want to use it to de-randomize the executable, they still need to guess 13 bits which has much more cost than trying to guess the executable directly because it have only 8 bits. The obtained address would be only useful if it is very close to one of the extremes of the range.

Therefore, although it seems paradoxical, assuming that the executable is the target, it is better to attack the executable directly than trying to leak an address of the HEAP.

3.4 Memory layout inheritance

It is widely known that children processes inherit/share the memory layout of the father. This is the expected behaviour, since children are a copy of the father. In most cases, the `fork()` is done as an intermediate step right before calling `exec()` to launch the new application; but there are some cases were multiple forks of the same process are running in parallel. For example, in the forking/pre-forking network servers model (used in Apache) and on general purpose application launchers like Zygote in Android or kinit in the KDE.

There are two straight forward consequences:

1. It is possible to perform **brute force attacks** to the ASLR, if the children are respawned (forked or cloned) when they are killed. In this case the attacker can do an exhaustive search of all possible addresses until the correct value is used. It worth to mention, that an attack similar to the byte-for-byte against the SSP [3] can also be done against the ASLR [4] when certain conditions are met.

2. A child process will know the position of future mappings of all other sibling processes, as far as the ASLR algorithm is deterministic (as it is the case in current implementations). This is specially

important in architectures where children are "independent applications". An example of this is Android where all applications are children (direct forks) of Zygote. Although the siblings might not call the same mapping sequence, a malicious sibling can predict future mmaps of any other sibling.

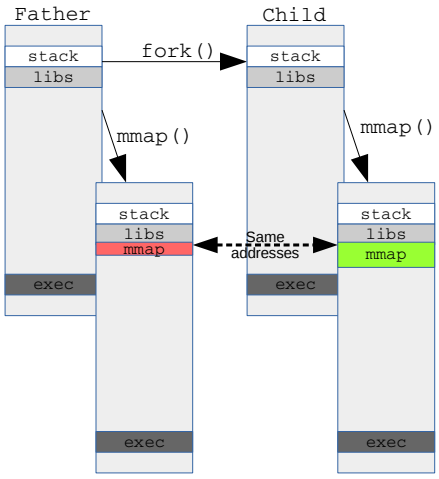


Figure 3.4 presents the memory layout of the father and a child process when the two processes request an mmap.

4 Design ASLR considerations

Abstract concepts are nice, but the real world is hard and very demanding. Here we will discuss the main factors that must drive an ASLR implementation. Note that future advances in processor architecture and software organization will modify this conditions, and so it will render some conclusions incorrect.

4.1 Performance

It is hard to find a security solution that does not have an impact on the performance (a nice exception is the NX/DEP mechanism). ASLR mainly relies on paging, and paging has a twofold impact on the performance: 1) temporal cost due to address resolution and 2) the size of the data structure (pagetable) used to perform the resolution. Both issues are normally related, the bigger the pagetable the longer is the time to walk-though it. As a rule of thumb, the smaller and more compact is the page table the better. One way to improve the performance is by increasing the granularity of the pages. Larger pages require smaller pagetables. Most processors support multiple page sizes (4Kb, 2Mb, 256Mb, etc.). Hugepages are mainly used by applications that has to manage large amounts of data, like databases[7] and scientific applications; but large libraries (as the Glibc) also benefit from Hugepages.

The randomness of the ASLR is achieved by scattering the objects along the memory. Unfortunately, a scattered layout requires more page directory entries, and so it is less efficient and memory consuming. Also, a random allocation may cause fragmentation because it may break the VMA in may small free chunks. Figure 4.1 shows the part of the addresses that are randomized in Linux 32 bits (x86) and 64 bit (x86_64). The lower part of the address is not randomized due to page alignment, and the upper part is not randomized in order to avoid fragmentation. Note that fragmentation is more relevant in 32 bits, and so, there are is less bits that can be randomized.

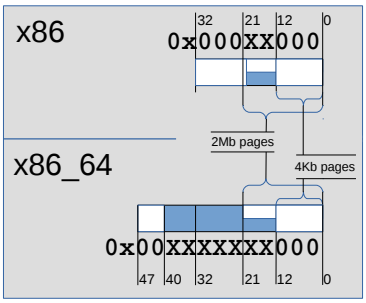


Figure 5: Part of the address that is randomized.

Linux 32bits has only 8 bits of entropy for libraries and mmaped objects, and when using huge pages, the page alignment sets the entropy to zero. On the other hand Linux 64, reserves less bits to avoid fragmentation than in 32 bits, which gives an effective

[7] It is almost mandatory when using the Oracle(r) database.

entropy of 28 bits. Even when huge pages are used, the available entropy is 19 bits, which is a fair number.

The more compact is the memory layout the better. A compact layout has a smaller pagetable (needs less middle page table entries and so less memory), which in turn uses less L2 cache. Obviously, if all the objects are located side by side, they will have total correlation between them. The problem with the vDSO in 64 bits [CVE-2014-9585] (only 11-bits were used to randomize the vDSO) was the consequence of trying to allocate it in the same middle page entry that the use used for the stack. We will see it later.

4.2 Alignment issues

Paging is the underlying mechanism used to implement ASLR. All pages must be aligned to page boundaries due to hardware requirements. Most memory operations provided by the OS (`mmap()`, `mprotect()`, `mlock()`,...) are required to work with page aligned addresses. Therefore, the lower bits of most objects are not random.

The page alignment limitation is augmented when the application uses huge pages. Rather than 4Kb or 8Kb per page, some processors can work larger pages (2Mb on x86), which both reduces the size of the page table and reduces the number of TLB entries.

It is called *sub-page* randomization when the page bits are also randomized. Linux and PaX uses sub-page randomization in the main stack[8], and PaX also uses it in the Heap and Arguments.

4.3 Growable objects

In ancient times, the Stack was located at higher addresses in order to have enough room to grow downwards. On the other side on the memory, we had the heap, which grows upward. This way, both objects could grow until they overlap each other. It was a simple, smart and effective solution.

But the `mmap()` facility introduced a new problem: objects created by the kernel shall be placed somewhere between the Stack and the Heap, fragmenting the nice and clean central memory space. Therefore, the kernel has to define the maximum size of the Stack when the process is initialized to define a new memory area to place mmap objects.

The `setrlimit()` system call can be used to change the size of the Stack (among other process resources). This feature allows us to control the maximum size of the Stack per application. By default Linux limits the Stack to 8MB. Other applications, like Oracle(r) data base, advises to use 10MB as a stack limit. We can say that the Stack of most processes of Linux is actually limited to a few mega bytes.

The other growable object is the Heap. The Heap is used by the application to allocate dynamic memory (`malloc()/free()`), the algorithms that implement those functions are generically called DSA, Dynamic Storage Allocator. When the DSA exhaust the memory pool used to attend `malloc()` requests, there are two forms to increase the memory pool: 1) expand existing pool by calling the `brk()` syscall and 2) request a new block of memory `mmap()` syscall.

The glib allocator, as well (dlmalloc, jemalloc, tcmalloc, tlsf, etc.) can work with multiple pools, which renders obsolete the `brk()` system call. In fact, the man page of brk saysin the CONFORMING TO section: "4.3BSD; SUSv1, marked LEGACY in SUSv2, removed in POSIX.1-2001."

Therefore, growable objects are no longer growable, i.e. they can be bounded. This observation opens the possibility to design more "aggressive", but compatible, ASLR designs.

4.4 Fragmentation

There are two issues related to the efficient use of the memory space: 1) The largest object that can be mapped and 2) The unusable gaps between the already allocated objects which can result in a out of memory.

The solution to both issues has been to limit the amount of virtual memory used by the ASLR. It seems that the randomness added to the position of the objects is wasted memory. And although this is mainly true, it is possible to have both, use the full VMA to allocate objects and at the same time allow large blocks allocations. Section 5 details how ASLR-NG solves this issues.

[8] The Stacks of the threads don't have sub-page randomization.

4.5 Backward compatibility

Some old applications rely on a very specific memory layout. For example, libc5 requires that heap to be right after the end of the `.bss` segment. Other applications assume a specific layout and then use (or abuse) the `MAP_FIXED` flag when requesting memory, which may cause collisions.

Linux provides the `personality()` syscall which, among other features, can set the memory layout for a given process as showed in table 1.

Personality	Description
`ADDR_NO_RANDOMIZE`	Disables randomization of the VMA
`ADDR_COMPAT_LAYOUT`	Changes the way virtual memory is allocated
`ADDR_LIMIT_32BIT`	Limit the virtual memory to 4GB
`ADDR_LIMIT_3GB`	Specifies that processes should use a maximum of 3GB

Table 1: Personality flags that have influence in the VMA.

The way the kernel applies the ASLR to all the processes is controlled by the `randomize_va_space` kernel parameter. Table 2 shows the possible values and their meanings. In current Linux versions the default is 2.

Value	Meaning
0	The ASLR is completely disabled
1	All randomized but the HEAP is loaded right after the executable
2	All objects are randomized

Table 2: /proc/sys/kernel/randomize_va_space values and their meanings.

5 ASLR Next Generation

Two main ideas have enabled us to design a new ASLR.

First we realized that growable objects (Stack and Heap) are no longer "growable", and can be handled and bounded or limited object without breaking backward compatibility. If we bound (limit the size of) the heap and the main stack (as indeed it is already happening now in practice), then it is possible to use the full range of VMA to allocate the objects. This is because the objects can not collide each other when growing.

But using the full VMA to randomize every single object may cause excessive fragmentation. This introduce basically two problems; 1) the fragmentation itself which prevents from guarantee large mmap requests, 2) the pagetable size is highly increased causing other effects like increasing the temporal cost (more TLB entries are needed, less entries are cached, etc.). The second core idea was how to extend the idea that we already used to address the Bulldozer issue (see appendix A.1.3) to avoid fragmentation without sacrificing entropy.

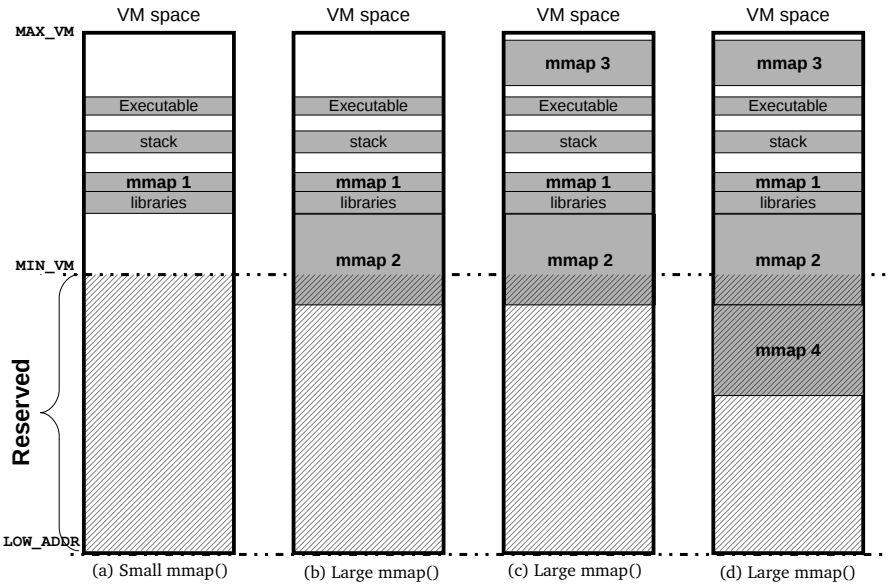
5.1 Addressing the fragmentation issue

We have solved the fragmentation problem by randomizing objects in two phases. When the process is initialized, the upper or the lower half side of the VMA is chosen at random (one random bit); and then during the execution of the process, the allocation algorithm uses the selected part to randomly allocate the new objects. Figure 5.1 sketches the solution.

This two phase randomization method provides both full range randomization (from the outside point of view) and at the same time half of the VMA is reserved to attend large requests. With this simple and smart solution, we have almost 20 random bits on 32-bit systems. Remember that 4Kb pages are aligned to 12 bits, and so, it limits the maximum randomization that can be obtained using the paging mechanism.

Once the fragmentation is no longer a problem and there is not growable objects, then we place any object at will. As a result, each object can be randomized independently, which in turn removes the correlation between them. This allocation strategy removes the correlation and inheritance weaknesses.

Another area of improvement is to add more random bits to the lower part of the addresses: sub-page randomization. This can be transparently done on objects that contain data, as the stack (both main



and thread stacks), the heap, the ARGV[9], and in some cases, the anonymous mmaps used to store data. The sub-page randomization can be transparently provided by the kernel to the stack and the heap (more precisely, the brk), but it is better implemented in the library for other type of objects.

5.2 Limiting the pagetable size

Another problem that must be addressed is the size of the pagetable. Even limiting the VMA used to randomize every object and mmap request, the pagetable size may be enormous, compared with current ASLR designs.

Since the balance between performance and security is not well defined and it is subject to subjective interpretation, we have designed the new ASLR-NG algorithm to be easily configurable. We have defined four operation modes:

Concentrated: All the objects are placed side by side in a fully random position. All objects are correlated. A compact layout is more efficient (smaller pagetable, and faster access) than a spread one. Note that the performance loss is not due to the use of the full VM space to randomize the objects, but it is because the maps are not grouped together.

Conservative: Four independent zones are defined: stack, heap, exec and libs/mmaps. This is the closest mode to current Linux and PaX ASLRs.

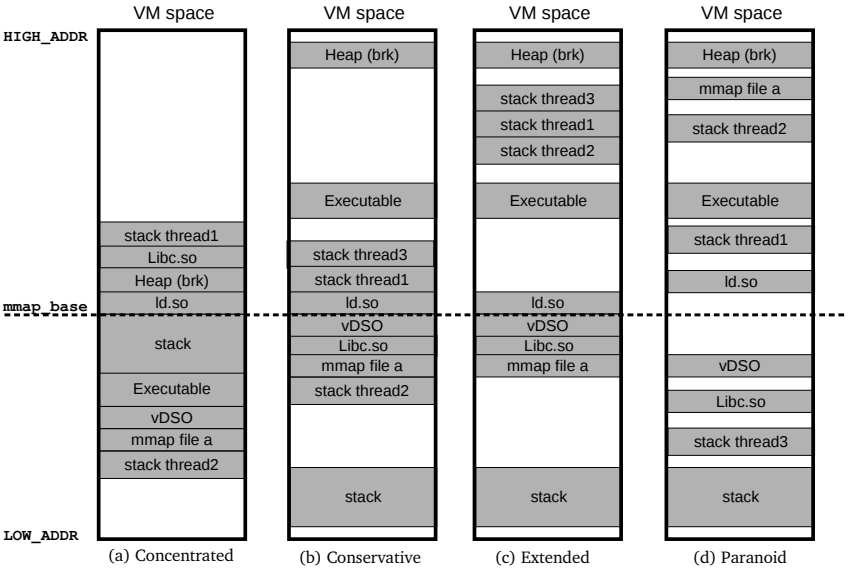
Extended: An extension of the conservative, where objects of the same type are grouped by zones/areas of memory. For example, the stacks of the threads are independently randomized from other kind of objects but side by the side mapped each other, also the hugepages, anonymous maps, etc.

Paranoid: Each object is independently randomized.

It worth to mention that all modes can provide the maximum possible entropy (as far as the reserved memory is 50%). In other words, the paranoid mode does not provides more (absolute) entropy bits for an object than the concentrated mode. The user or the administrator is encouraged to select a more aggressive ASLR-NG mode only when the correlation (see section 3.3) can be an issue. On the other side, for example, by selecting the concentrated mode, it provides the maximum possible entropy and at the same time save pagetable size (even comparing with current ASLRs implementations).

These modes can be set at system level, via a kernel parameter `/proc/sys/vm/aslrng_mode`, and on a per process basis using the `personality()` system call.

[9]The ARGV object is the block of memory that contains the parameters and environmental variables received by the process.



(a) Concentrated (b) Conservative (c) Extended (d) Paranoid

Figure 6: ASLR-NG: Profile mode examples.

5.3 ASLR-NG evaluation

Next (figure 7) is compares the entropy of ASLR-NG, PaX ASLR for 64-bit systems. ASLR-NG has been configured to work in conservative mode and with the 50% of pre-reserved virtual memory.

Update Trials x sec: 1000	ASLR-NG 4.5.0		PaX 3.14.21	
Object	**Entropy**	**Time**	**Entropy**	**Time**
Arguments	47.0	$4 \cdot 10^3$ year	39.0	17 year
Main_stack	43.0	278 year	35.0	1 year
HEAP	43.0	278 year	35.0	1 year
MAP_SHARED	35.0	1 year	28.5	4 days
Glibc	35.0	1 year	28.5	4 days
VDSO	35.0	1 year	28.5	4 days
EXEC	35.0	1 year	27.0	1 days
Dynamic_Loader	35.0	1 year	28.5	4 days
MAP_HUGETLB	26.0	18 hours	19.5	12 mins

(a) 64-bit system

Update Trials x sec: 1000	ASLR-NG 4.5.0		PaX 3.14.21	
Object	**Entropy**	**Time**	**Entropy**	**Time**
Arguments	31.6	37 days	27.0	1 days
Main_stack	27.6	2 days	23.0	2 hours
HEAP	27.6	2 days	23.4	3 hours
MAP_SHARED	19.6	13 mins	15.7	53 secs
Glibc	19.6	13 mins	15.7	53 secs
VDSO	19.6	13 mins	15.7	53 secs
EXEC	19.6	13 mins	15.0	32 secs
Dynamic_Loader	19.6	13 mins	15.7	53 secs
MAP_HUGETLB	10.6	1 secs	5.7	0 secs

(b) 32-bit System

Figure 7: ASRL-NG (conservative and 50% reserved) vs Pax.

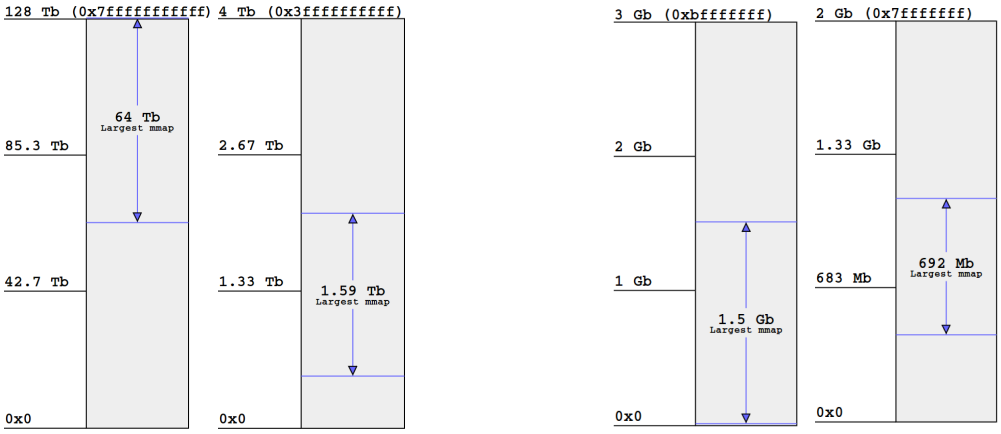
As showed in figure 7, ASLR-NG out performs PaX in all objects. Using the sub-page randomization it is possible to have the 100% of entropy (47 bits) on the Arguments object. And the full VMA (35, which is 47 bits minus 12 page bits) in all objects but Hugepages.

Besides the entropy, the other critical parameter of an ASLR design is the fragmentation. Figure 8 shows the largest mmap that can be requested in PaX and ASLR-NG.

PaX limits the largest mmap request to 692 MB in 32-bit systems which could prevent the execution of some applications. For instance, applications that need to map files of 1GB. In contrast, ASLR-NG allows much more free consecutive space than PaX and Linux. The maximum size of a mmap request in ASLR-NG by default is similar to the allowed in Linux, that is 1.5GB (50% of the VMA), but ASLR-NG provides a mechanism to tune this value in order to increase or decrease the "pre-reserved" area. For example by setting the "pre-reserved" area to 90%, ASLR-NG provides more entropy than Linux and PaX by only using 300MB of VA and at the same times has a free consecutive space of 2.7G.

PaX restricts all objects to be in the lowest 4TB of the VMA out of the 128TB. As far as we know, there is no performance or compatibility restriction that justify that low limit. On the other hand ASLR-NG and Vanilla Linux can use the full VMA. Vanilla Linux can allocate the largest block. But ASLR-NG can be configured to pre-reserve more that the 50% in order to have less fragmentation than vanilla allowing allocations larger than Vanilla and much more entropy.

When ASLR-NG is configured with 60% of pre-reserved memory, the resulting entropy does not get too degraded because still a very large potion of the VMA is used, as can bee seen in the figure 9.



(a) 64-bit ASLR-NG (b) 64-bit PaX (c) 32-bit ASLR-NG (d) 32-bit PaX

Figure 8: Largest usable mmapped object on 32 and 64 bits.

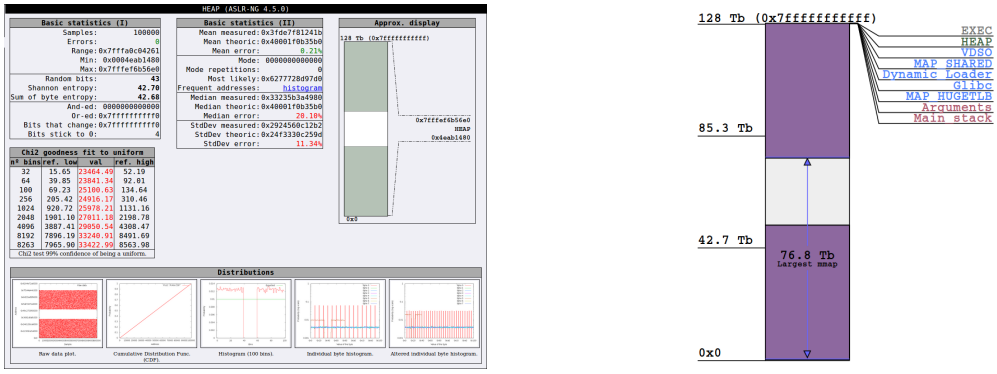


Figure 9: ASRL-NG with 60% of pre-reserved memory.

.

As can be seen in the figure, the largest mmap (76.8TB) can be located at lower or upper addresses with a probability of 50%, it will be located in the opposite side where ASLR-NG allocates objects. It is also interesting to see that an attacker can not discard any of the purple zones, but at the same time those areas are also reserved to attend large blocks.

6 ASLRA: ASLR Analyzer

In order to measure the effective entropy of each object, the Grsecurity developed a tool called `paxtest`[10]. It is a easy to use tool, but it gives very little information, and in some cases the **results are incorrect**.

The following issues of the paxtest tool forced us to develop a new tool to analyze the ASLR:

1. The quality of the entropy is measured as a single statistical parameter: the random bits, which is measured using an adhoc heuristic. This heuristic works well when the distribution is uniform and its range is a power of two, aligned to a power of two. Otherwise, it is n "educated guess" of the real entropy. See Appendix A.3.

2. It performs only 1000 tests. But in order to obtain statistically significance, it is necessary to collect millions of samples, specially on 64-bit systems.

3. The address of the main executable is not correctly read. Rather than the address of the main function, the code gets the address of the library. A critical value (the randomization of the executable) is not correctly measured. See Appendix A.2.

[10]Paxtest is available in most Linux distros ($ sudo apt-get install paxtest).

We have developed a more complete analysis tool called ASLRA (ASLR Analyzer), which is still under active development. Most of the results presented on this paper has been obtained with this tool. will released it soon.

The main features of ASLRA are:

- An efficient sampler tool, which is able to perform millions of samples in a few minutes.
- Complete statistical analysis: mean, mode, std. distribution, CDF, histograms, etc.
- Three estimators of entropy: Shannon, Shannon at byte level and "flipping bits".
- Graphical representation of the VMA layout.
- Compares and summarizes multiple systems.
- Chi2 test fit for uniform distribution, very effective to detect biased behavior.
- Measurement of the largest mmap.
- Correlation between objects (still under development).
- The output is presented in interactive HTML/SVG.

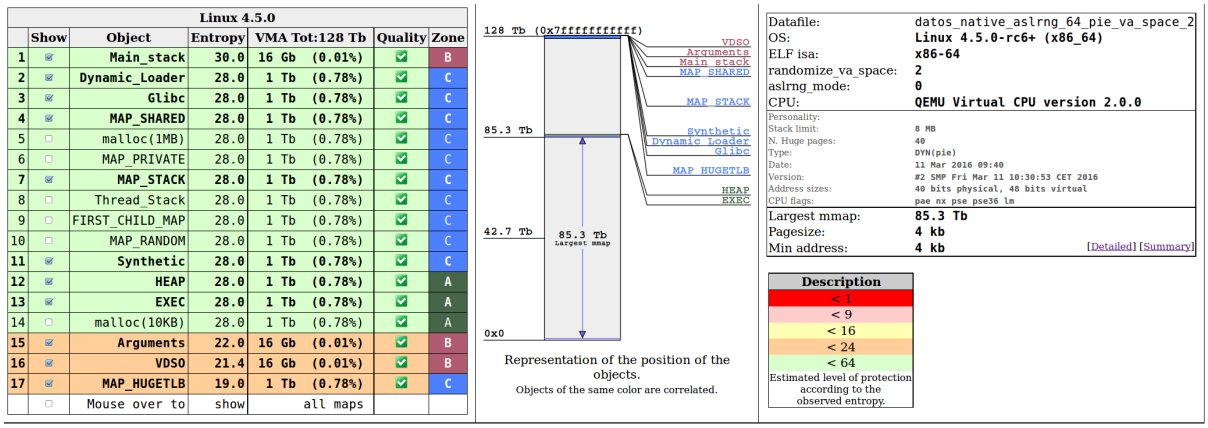


Figure 10: Summary output of the ASLRA tool for a Vanilla Linux system.

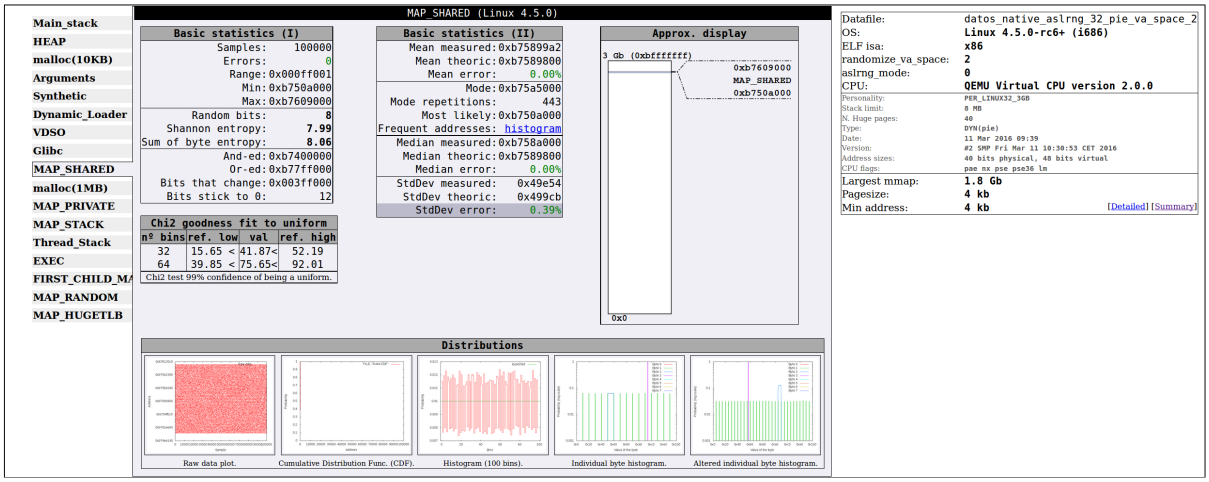


Figure 11: Complete statistical information about single object.

The correlation coefficient corresponds to the Pearson's or PCC coefficient, which measures the "linear" correlation between the positions of two objects. For our purposes, it is not the most useful

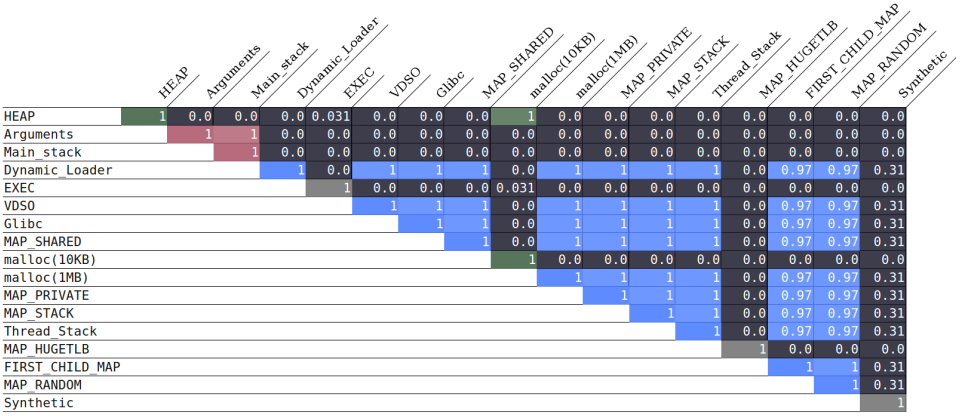


Figure 12: Simple statistical correlation between objects.

statistical parameter because it gives more weight to the upper address bits than to the lower ones. Higher bits are more significant than lower bits, but for our purposes any bit shall have the same impact. A more accurate statistical parameters are "mutual information" and "conditional entropy", but those parameters are hard to estimate when the range is much larger than the number of samples.

7 Conclusions

ASLR is a very powerful mitigation technique, which in many cases it is the last line of defense. Unfortunately, it is very system and implementation dependent. On 32 bit systems, the protection provided by current ASLR implementations can be questioned (even using the best implementations available) because of their low entropy.

This white paper tries to shows the main limitations and weaknesses of current Linux and PaX implementations. Thanks to the huge VM space on 64 bit systems it is *relatively* easy to design a robust ASLR. There is plenty of room to randomize the addresses without causing fragmentation. But 32bits seemed to be hopeless to have a reasonable ASLR.

We presented the ASLR-NG a new ASLR that both, provides the maximum entropy from the available VMA and introduces a novel solution for the ASLR fragmentation. This new design is specially appealing to 32-bit systems because the small VMA available. Those small low power, low consumption devices are the building blocks for Internet of Things.

We have developed ASLRA, a tool to analyze the quality of the ASLR. This tool shows that ASLR-NG outperforms PaX ASLR in all aspects.

A Appendixes

A.1 Vulnerabilities found in Linux and PaX ASLR implementations

During the study and the development of the proposed ASLR-NG, we found several weakness as described in section 3 and some vulnerabilities. Following is a brief of them.

A.1.1 CVE-2015-1593 - Linux ASLR integer overflow: Reducing stack entropy by four

The Linux ASLR implementation for versions prior to 3.19-rc3 did not properly randomized the stack for processes on some 64 bit architectures due to an **integer overflow**. Affected systems have reduced the stack entropy of the processes by four [5]. This vulnerability did not affect PaX.

Two bits are dropped due to an integer overflow, from 2^{30} to 2^{28}, which reduces the stack entropy by four. The number of possible places to allocate the stack due to integer overflow is approximately 268 millions instead of 1 billion. Maybe for this reason this bug has remained hidden for so long.

A.1.2 Linux ASLR mmap weakness: Reducing entropy by half

The vulnerability affects Linux ASLR [6]. The issue is that the mmap base address for processes is not properly randomized on some architectures due to an improper **bit-mask manipulation**. Affected systems have reduced the mmap area entropy of the processes by half.

Although only one bit is dropped, from 2^{18}(131072) to 2^{17}(262144), the range of mmap base area is reduced by half.

A.1.3 AMD Bulldozer Linux ASLR weakness: Reducing entropy by 87.5%

We found a security issue affecting Linux and PaX ASLR in the AMD's Bulldozer processor [7] family. The issue affects to all Linux process even if they are not using shared libraries (statically compiled).

The problem appears because some mmapped objects (VDSO, libraries, etc.) are poorly randomized in an attempt to avoid cache aliasing penalties in this specific processor family.

Affected systems have reduced the mmapped files entropy by eight, which is specially dangerous in 32-bit systems because of the low entropy available.

A.1.4 Offset2lib: Bypassing the Full ASLR on 64 bit GNU/Linux in less than 1 sec.

Offset2lib [4] showed how the total correlation weaknesses (between the executable and libraries) could be exploited to bypass the ASLR. Linux located in the mmap area the image of the executable when it was PIE compiled, which resulted in (total correlation).

The PoC bypassed the three most widely adopted and effective protection techniques: No-eXecutable bit (NX), address space layout randomization (ASLR) and stack smashing protector (SSP); using the byte-for-byte strategy against both the SSP and the ASLR.

A.2 Patch for the paxtest tool

Here is the patch to fix the problem when reading the address of the main function, which is used to estimate the entropy of the executable. Unfortunately, due to the inline optimization of the compiler (default option when -O2 is used), the `foo()` function gets inlined into the `main()` function, and thus the value returned by `__builtin_return_address(0)` is the address of the library function which called `main()` rather than the address of main.

```
#include <stdio.h>
#include <stdlib.h>

void foo(void)
{
       printf( "%p\n", __builtin_return_address(0) );

}

int main( int argc, char *argv[] )
{
       foo();
       exit(0);
}
```

Listing 1: Code to read the address of main. File: `getmain.c`

```
diff -rupN paxtest-0.9.13/getmain.c paxtest-0.9.13.fixed/getmain.c
--- paxtest-0.9.13/getmain.c 2015-03-06 11:03:25.218686546 +0100
+++ paxtest-0.9.13.fixed/getmain.c  2015-03-06 11:16:41.614669807 +0100
@@ -7,10 +7,9 @@
 #include <stdio.h>
 #include <stdlib.h>

-void foo(void)
+void __attribute__ ((noinline)) foo(void)
 {
       printf( "%p\n", __builtin_return_address(0) );
```

```
-
 }

 int main( int argc, char *argv[] )
```

Listing 2: Patch to fix the random address of the main() function.

As a consequence, the measured entropy of the executable is incorrect. And vanilla Linux has more entropy than PaX on the EXECS.

A.3 How paxtest estimates the entropy

Entropy is hard to measure, but we believe that the way it is done in the paxtest (see figure 3) is a nice try but not perfect.

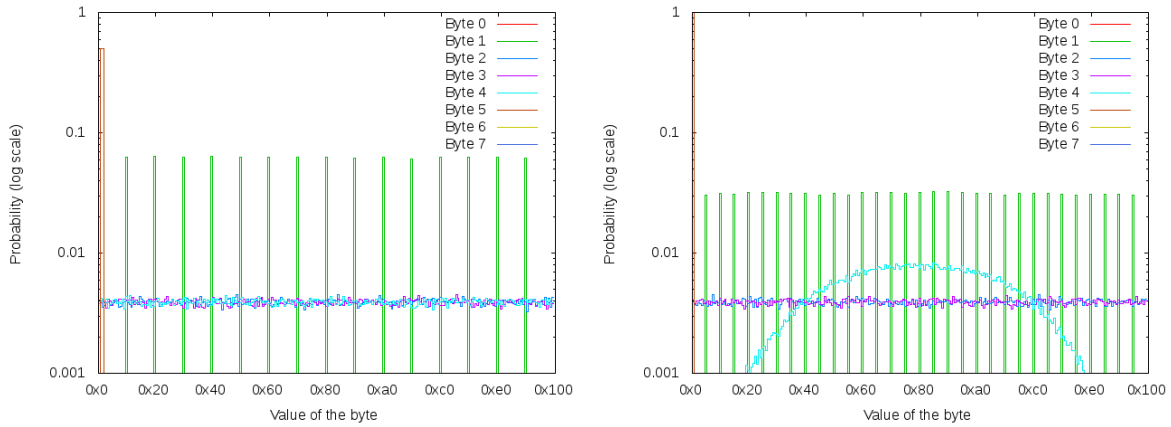
```
. . .
and = ~0L;
or = 0L;
for( i = 0; i < COUNT; i++ ) {
  fscanf( fp, "%lx", &tmp );
  results[i] = tmp;
  and &= tmp;
  or |= tmp;
}
tmp = and ^ ~or;
tmp = or & ~tmp;
bits = 0;
for (i = 0; i < sizeof(quality)/sizeof(quality[0]); i++) {
  if (!(tmp & (1UL << i)))
  continue;
  for (x = 0; x < COUNT; x++) {
    if (results[x] & (1UL << i))
    quality[i] += 1;
  }
  if (quality[i] <= ((COUNT * 35) / 100) || quality[i] >= ((COUNT * 65) / 100))
  weak_bits++;
}
while( tmp != 0 ) {
  bits += (tmp%2);
  tmp >>= 1;
}

printf( "%d quality bits (guessed)\n", bits - weak_bits);
```

Listing 3: Code of the paxtest tool that estimates the entropy (quality bits).

It counts the number of bits that change, `bits`, and subtracts `weak_bits`, those bits that do not change too often (less than 35% and more than 65%). Each bit is analyzed independently from the rest of the bits of the address. Unfortunately, it produces an incorrect result when the distribution is non-uniform but symmetric and centered in a power of two. An example of this scenario is the libraries and mmaped objects in PaX 64-bits.

The best way to observe this effect of bit and byte "folding" is by plotting the probability of each individual byte of the address, and then do the same plot but dividing each address by a factor (in our case we have used 2, but it may be better to use a larger prime). The bits or bytes of the resulting address are not symmetrically folded and real distribution of the original variable can be appreciated, as can be seen in following pair of plots A.3.

As can be seen, the 4^{th} byte is not uniform along the 0..255 range. A graphical explanation of the aliasing or folding is shown in the next figure, where the distribution is triangular with range 512 and centered at 255. In this scenario, each individual byte follows a perfect uniform distribution.



(a) Probability of individual bytes of addresses.

(b) Probability of individual bytes of addr×2.

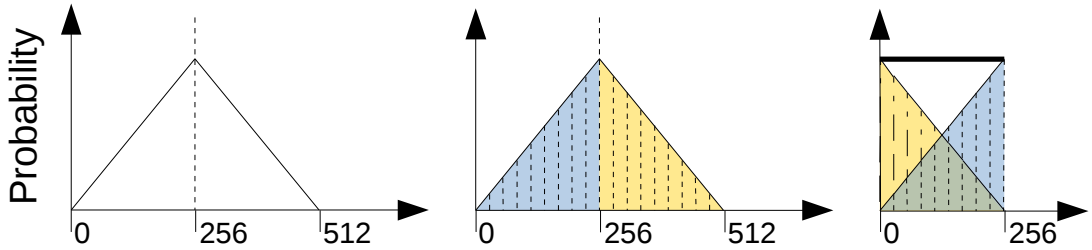


Figure 13: Graphical example of the byte folding effect.

References

[1] F. J. Serna. (2015, February) glibc getaddrinfo stack-based buffer overflow. [Online]. Available: https://security.googleblog.com/2016/02/cve-2015-7547-glibc-getaddrinfo-stack.html

[2] M. S. Johnstone and P. R. Wilson, "The memory fragmentation problem: Solved," in *Proceedings of the First International Symposium on Memory Management, ACM.* Press, 1998.

[3] A. 'pi3' Zabrocki, "Scraps of notes on remote stack overflow exploitation," November 2010. [Online]. Available: http://www.phrack.org/issues.html?issue=67&id=13#article

[4] H. Marco-Gisbert and I. Ripoll, "On the effectiveness of full-aslr on 64-bit linux," in *In-depth security conference, DeepSec*, November 2014. [Online]. Available: http://cybersecurity.upv.es/attacks/offset2lib/offset2lib-paper.pdf

[5] ——. (2015, January) CVE-2015-1593 - Linux ASLR integer overflow: Reducing stack entropy by four. [Online]. Available: http://hmarco.org/bugs/linux-ASLR-integer-overflow.html

[6] ——. (2015, January) Linux ASLR mmap weakness: Reducing entropy by half. [Online]. Available: http://hmarco.org/bugs/linux-ASLR-reducing-mmap-by-half.html

[7] ——. (2015, March) AMD Bulldozer Linux ASLR weakness: Reducing entropy by 87.5%. http://hmarco.org/bugs/AMD-Bulldozer-linux-ASLR-weakness-reducing-mmaped-files-by-eight.html.