1. Design

The goal of Address Space Layout Randomization is to introduce randomness into addresses used by a given task. This will make a class of exploit techniques fail with a quantifiable probability and also allow their detection since failed attempts will most likely crash the attacked task.

To help understand the ideas behind ASLR, let's look at an example task and its address space: we made a copy of /bin/cat into /tmp/cat then disabled all PaX features on it and executed "/tmp/cat /proc/self/maps". The [x] marks are not part of the original output, we use them to refer to the various lines in the explanation (note that the VMMIRROR document contains more examples with various PaX features active).

```
 [1] 08048000-0804a000 R+Xp 00000000 00:0b 812        /tmp/cat
 [2] 0804a000-0804b000 RW+p 00002000 00:0b 812        /tmp/cat
 [3] 40000000-40015000 R+Xp 00000000 03:07 110818     /lib/ld-2.2.5.so
 [4] 40015000-40016000 RW+p 00014000 03:07 110818     /lib/ld-2.2.5.so
 [5] 4001e000-40143000 R+Xp 00000000 03:07 106687     /lib/libc-2.2.5.so
 [6] 40143000-40149000 RW+p 00125000 03:07 106687     /lib/libc-2.2.5.so
 [7] 40149000-4014d000 RW+p 00000000 00:00 0
 [8] bfffe000-c0000000 RWXp fffff000 00:00 0
```

As we can see, /tmp/cat is a dynamically linked ELF executable, its address space contains several file mappings.

[1] and [2] correspond to the loadable ELF segments of /tmp/cat containing code and data (both initialized and uninitialized), respectively.

[3] and [4] represent the dynamic linker whereas [5], [6] and [7] are the segments of the C runtime library ([7] holds its uninitialized data that is big enough to not fit into the last page of [6]).

[8] is the stack which grows downwards.

There are other mappings as well that this simple example does not show us: the brk() managed heap that would directly follow [2] and various anonymous and file mappings that the task can create via mmap() and would be placed between [7] and [8] (unless an explicit mapping address outside this region was requested using the MAP_FIXED flag).

For our purposes all these possible mappings can be split into three groups:
- [1], [2] and the brk() managed heap following them,
- [3]-[7] and all the other mappings created by mmap(),
- [8], the stack.

The mappings in the first and last groups are established during execve() and do not move (only their size can change) whereas the mappings in the second group may come and go during the lifetime of the task. Since the base addresses used to map each group are not related to each other, we can apply different amount of randomization to each. This also has the benefit that whenever a given attack technique needs advance knowledge of addresses from more than group, the attacker will likely have to guess or brute force all entropies at once which further reduces the chances of success.

Let's analyze now the (side) effects of ASLR. For our purposes the most important effect is on the class of exploit techniques that need advance knowledge of certain addresses in the attacked task, such as the address of the current stack pointer or libraries. If there is no way to exploit a given bug to divulge information about the attacked task's randomized

address space layout then there is only one way left to exploit the bug: guessing or brute forcing the randomization.

Guessing occurs when the randomization applied to a task changes in every attacked task in an unpredictable manner. This means that the attacker cannot learn anything of future randomizations and has the same chance of succeeding in each attack attempt. Brute forcing occurs when the attacker can learn something about future randomizations and build that knowledge into his attack. In practice brute forcing can be applied to bugs that are in network daemons that fork() on each connection since fork() preserves the randomized layout, as opposed to execve() which replaces it with a new one. This distinction between the attack methods becomes meaningless if the system has monitoring and reaction mechanisms for program crashes because the reaction can then be triggered at such low levels that the two attack methods will have practically the same (im)probability to succeed.

To quantify the above statements about probability of success, let's first introduce a few variables:

  Rs: number of bits randomized in the stack area,
  Rm: number of bits randomized in the mmap() area,
  Rx: number of bits randomized in the main executable area,
  Ls: least significant randomized bit position in the stack area,
  Lm: least significant randomized bit position in the mmap() area,
  Lx: least significant randomized bit position in the main executable area,
  As: number of bits of stack randomness attacked in one attempt,
  Am: number of bits of mmap() randomness attacked in one attempt,
  Ax: number of bits of main executable randomness attacked in one attempt.

For example, for i386 we have Rs = 24, Rm = 16, Rx = 16, Ls = 4, Lm = 12 and Lx = 12 (e.g., the stack addresses have 24 bits of randomness in bit positions 4-27 leaving the least and most significant four bits unaffected by randomization). The number of attacked bits represents the fact that in a given situation more than one bit at a time can be attacked (obviously A <= R), e.g., by duplicating the attack payload multiple times in memory one can overcome the least significant bits of the randomization.

The probabilities of success within x number of attempts are given by the following formulae (for guessing and brute forcing, respectively):

  (1) Pg(x) = 1 - (1 - 2^-N)^x, 0 <= x
  (2) Pb(x) = x / 2^N, 0 <= x <= 2^N

  where N = Rs-As + Rm-Am + Rx-Ax, the number of randomized bits to find.

Based on the above the following tables summarize the probabilities of success as a function of how many bits are tried in one attempt and the number of attempts.

Pg(x)| x

N	1	4	16	64	256	2^10	2^14	2^18	2^20	2^24	2^32	2^40	2^56	2^64
1	0.50	0.94	~1	~1	~1	~1	~1	~1	~1	~1	~1	~1	~1	~1
2	0.25	0.68	0.99	~1	~1	~1	~1	~1	~1	~1	~1	~1	~1	~1
4	0.06	0.23	0.64	0.98	~1	~1	~1	~1	~1	~1	~1	~1	~1	~1
8	~0	0.02	0.06	0.22	0.63	0.98	~1	~1	~1	~1	~1	~1	~1	~1
16	~0	~0	~0	~0	~0	0.02	0.22	0.98	~1	~1	~1	~1	~1	~1
24	~0	~0	~0	~0	~0	~0	~0	0.02	0.06	0.63	~1	~1	~1	~1
32	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	0.63	~1	~1	~1
40	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	0.63	~1	~1

56	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	0.63	~1

Pb(x) | x

N	1	4	16	64	256	2^10	2^14	2^18	2^20	2^24	2^32	2^40	2^56	2^64
1	0.50													
2	0.25	1												
4	0.06	0.25	1											
8	~0	0.02	0.06	0.25	1									
16	~0	~0	~0	~0	~0	0.02	0.25							
24	~0	~0	~0	~0	~0	~0	~0	0.02	0.06	1				
32	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	1			
40	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	1		
56	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	1	

It is obvious that from the defense point of view the goal would be to make N as high as possible while keeping x as low as possible. Unfortunately N is not under control by the defense side (re-randomizing the address space at runtime is not feasible because part of the necessary relocation information is simply lost), but rather that of the nature of the bug and the attacker's exploit skills. What we know is that there has been very little research done and published on countering ASLR so far (e.g., it is unknown how certain real-life bugs such as stack or heap overflows can be used for information leaking in a general way). Reducing N is possible if an attacker can store multiple instances of the attack payload (e.g., stack frame chain if NOEXEC is active, or some shellcode when it is not) in the attacked task's address space. This would typically be possible by exploiting an overflow style bug where the attacker can fill a contiguous range of memory with data of his choice. As the size of this memory range grows above the value of L relevant to the given range, more and more randomized bits can be ignored in the attack payload. For example, to overcome all of R for a given range on i386, the attacker would have to send 256 MB of data, something that is not always possible (e.g., the stack typically has a maximum limit of 8 MB and grows to much less in practice).

It is also unknown how bugs that have been neglected so far can be used against ASLR, that is, bugs that give only read access (vs. write) to the attacker and were not considered as a serious security problem before but may now be used to help counter ASLR in an exploit attempt of another bug.

On the other hand however the defense side has quite some control over the value of x: whenever an attack attempt makes a wrong guess on the randomized bits, the attacked application will go into a state that will likely result in a crash and hence becomes detectable by the kernel. It is therefore a good strategy to use a crash detection and reaction mechanism together with ASLR (PaX itself contains no such mechanism).

The last set of side effects of ASLR is address space fragmentation and entropy pool exhaustion. Since randomization shifts entire ranges of memory, it will also randomly change the gaps between them (which were constant before). This in turn will change the maximum size of memory mappings that will fit in there and applications expecting to be able to create them will fail. Finally, ASLR increases the consumption of the system's entropy pool since every task creation (through the execve() system call) requires some bits of randomness to determine the new address space layout. Depending on the system's threat model however a given implementation can relax the requirements for the quality of this entropy. In particular, if only remote attacks are considered, then ASLR does not need cryptographically secure random bits as a remote attacker cannot observe them (or if he can, he does not need to care about ASLR at all).

2. Implementation

PaX can apply ASLR to tasks that are created from ELF executables and use ELF libraries. The randomized layout is determined at task creation time in the load_elf_binary() function in fs/binfmt_elf.c where three per task (or more precisely, mm_struct) variables are initialized with random numbers: delta_exec, delta_mmap and delta_stack.

The following list specifies which ASLR feature affects which part of the task address space layout (they are discussed in detail in separate documents):

```
RANDEXEC/RANDMMAP (delta_exec)  - main executable code/data/bss segments
RANDEXEC/RANDMMAP (delta_exec)  - brk() managed memory (heap)
RANDMMAP          (delta_mmap)  - mmap() managed memory (libraries, heap,
                                  thread stacks, shared memory)
RANDUSTACK        (delta_stack) - user stack
RANDKSTACK                      - kernel stack (not part of the task's
                                  address space)
```

The main executable and the brk() managed heap can be randomized in two different ways depending on the file format of the main executable. If it is an ET_EXEC ELF file, then RANDEXEC can be applied to it, if it is an ET_DYN ELF file then RANDMMAP.