Limitations of Absolute Address Randomization

It's not possible to achieve complete randomness in ASR. Some memory regions, such as libraries, may be required to be aligned on page boundaries, thus making their lower-order bits predictable. The attacker can corrupt these bits in a predictable way.

Brute force attacks are still possible in which the attacker keeps trying different values till she succeeds. A successful attack was carried out several years ago by Shacham et al on 16 bit randomness of ASR implementation (called ASLR) by the PaX project. The attack succeeded by trying 32K values in around 2 minutes. This attack shows that an E-commerce server cannot rely solely on ASR as a defense. At the minimum, contingency measures need to be taken when repeated crashes are observed.

Also note that if you consider an Internet-wide worm, ASR offers only limited protection. In this case, the worm may try to attack millions of computers on the Internet at the same time. With 16-bit randomization, the attack will succeed on 1 of 64K computers on the very first attempt. Thus, in this scenario, even a contingency measure based on repeated crashes does not help. It is likely that the spread of the worm will be slowed down, as it is going to take many more attempts to infiltrate into a system due to the use of ASR.

Partial Pointer Overwrite: The concept of PPO is to modify the least significant bytes. In x86 it is possible to change least significant 1-2 bytes of a pointer. If the randomness is contained in the upper 16 bits of a pointer, this means that the attacker can predict the lower 16 bits. For instance, it is easy to mount return-to-libc attacks using this approach. The RA on the stack is going to correspond the the location of some instruction in a library (or the executable). Since the relative distance between this instruction and other code in the library, he can simply modify the least significant 2-bytes of the RA so that it will now point to the code of attackers choice that resides within the same library.

Note that the attacker has to rely on other vulnerabilities than strcpy for PPOs since a strcpy will write a null-byte following attacker-provided bytes, which will corrupt the next byte in the pointer.

Many integer overflow vulnerabilities result in an out-of-bounds access, which involves a base address and an offset. It is the offset that is involved in overflow, while the base address remains fixed in the program. So, in this case, we essentially have a relative address attack, which AAR does not protect against. However, sometimes the offsets can be very large, resulting in an address that goes past the end of a memory region (like static memory) into another region (like the heap). In this case, AAR can help since the distance between two regions becomes random with AAR.

Information Leakage Attacks: An address value escapes and gets back to the attacker, e.g. a server can send back some data to the client, who is actually an attacker. Due to a bug in the system, the server may send more data than is intended, for instance, it may malloc a buffer of 64-bytes, fill it with 32-bytes of data, but then send all 64- bytes back. The uninitialized 32 bytes may contain pointer values. By comparing these values with the same values on an instance of the victim program on attacker's own machine, she can identify the random offset used by the AAR scheme. After this she can correctly guess all the addresses of all data and data locations at the server.

ASR is still not very widely deployed. As a result the feasibility and ease of information leakage attack is not well known. It may very well turn out that information leakage vulnerabilities are common – if so, AAR will not provide any protection at all.

One fundamental deficiency of AAR is that the mechanism relies on a single random number. When the number is leaked, the entire system becomes compromised. A proposal can be to employ various random bases so that even some of the bases are compromised; the rest of the system stays trusted.

More generally, AAR is not effective against data attacks or other attacks described above that rely only on relative distances. This motivates the relative address randomization techniques described below.

Relative address randomization

An improvement over the Absolute Address Space Randomization is the Relative Address Space Randomization or RAR. In RAR, the relative addresses of all individual code and data objects are randomized. RAR focuses mainly on static and stack variables, and code. It does not focus on heap since relative address randomization in heap is somewhat easier to achieve --- simply change the malloc library.

The relative distance between different functions is also randomized. Code transformation can be used to permute the relative order of routines in memory. Changing the relative address of existing functions thwarts return-to-libc attacks.

An improvement to this kind of randomization is achieved using write-protected memory pages between two pages to prevent overflow. If an overflow occurs in one of the segments that flow beyond its boundary, it will go into the protected page and a memory fault will arise. Implementing this improvement entails space overhead. Another price is that each page that needs to be protected requires a system call. If you have to introduce 100K pages that are protected, then it will entail significant startup overhead.

A compiler can allocate two static variables in any order as the C language doesn't place any restrictions or norms about this. As a result an overflow in one variable cascades to the other. With RAR, each time the program is run, the relative positions of the variables are different so it can never be predicted if the overflow in a particular variable will reach to some other particular variable, which is the basic requirement behind most attacks.

Side discussion: the Propolice defense changes the relative order of some local variable, but this is done in a specific way that maximizes the effectiveness of the canary. In particular, there is absolutely no randomness involved, and the relative distances can be predicted.

Compile time randomization is not very useful. This is because, with current distribution models, the same binary is distributed and installed on all machines. Thus, a compile-time randomization won't contribute to any diversity across the population. Even if the installation model changes so that each software download is a differently compiled binary, there is still the problem that the resulting binary would have the exact same randomization each time it is run. This will mean that attackers can monotonically gain information with each attack attempt. In contrast, if the randomization is determined at load time or runtime, each execution is different, and an attack during one run will not yield information regarding randomization that is useful for another run.

With all the security mechanism in place, some random attacks may still succeed but it's not possible for an attacker to launch a predictable attack (unless the attack is repeated many, many times.)

For security against stack-smashing attacks, there is another mechanism that can be used: split the stack into into two parts. One part contains the values that are usual targets for overflows, which are mainly control data like Base Pointer and return address while the other part contains variables that are prone to overflow, e.g., arrays. The two stacks can be back to back, possibly with a protected memory page inbetween to prevent any possibility of overflow from one stack to another. Since the first stack does not contain any object that can be involved in an overflow, the values on that stack are safe from being corrupted by a buffer overflow. (The RAR technique described below uses this dual-stack approach, and randomizes the order of variables on the second stack, while performing no randomization on the first stack.)

Benefits of RAR:

Higher entropy: The attacker needs much more information to break the RAR as the randomization is done at a finer granularity. Up to 28 bits of address can be randomized, as against 16 bits of AAR.

Information leakage attacks are not effective --- even if some information is leaked regarding the location of one object, this does not help the attacker know the location of other objects in memory.

For heap overflow attacks, the attacker needs to guess 2 pointer values: first, the location of a function pointer that he wishes to change, and second, the location of the code that this function pointer should be pointed to. With basic AAR, it is typically the case that once the first pointer value is correctly guessed, the second one can be guessed as well, since every thing is based on a single random value. With RAR, this is no longer the case.

Runtime randomization of each variable requires extra information as the binaries have no information about variable boundaries. In the RAR technique described here, additional information is encoded into the binary that helps achieve relative randomization. The details can be found in this paper.

DATA SPACE RANDOMIZATION:

The data space randomization focuses on the randomization of interpretation of data. The main concept behind DSR is to randomize the effect of overflows so that the effects of data corruption become non-deterministic.

In the context of DSR, it is important to realize that data internal to a program can be represented internal to the program in any way that the program chooses. For data that is involved in external communication, this is not true.

(See the slides used in lectures for more information. You can also look at this paper for a more detailed presentation.)

One thing to notice is that for all randomization problems, we essentially consider weak adversaries, i.e. those who cannot run a program on the system. If an adversary can already run code, then randomization does not provide much protection, as the running code can scan the memory to identify the random values being used.

Benefits of DSR:

Provides greater entropy as it makes 32 bit randomization possible. It can use different masks for different variables so each overflow can be immediately detected at the time the data is examined. It protects all data, not just pointers, and it is effective against relative address attack as well as absolute address attack. It can detect intra-structure overflows which are difficult to detect using any other mechanism.

Intra-structure overflow:

In a structure, each variable is saved at a location that is a multiple of its size and the leftover space is left blank.

```
Struct    {
          char a;
          short b;
          double f;
          int d;
  }
```

Base +0	a
	Empty space
Base +2	b
	Empty space
Base +8	f
Base +16	d

A processor architecture (and possibly, a compiler) defines certain aspects such as sizes of primitive data types such as short and int are defined, as are their alignment requirements. For instance, on the x86 architecture, integers are 32-bit while shorts are 16-bits. In addition, integer variables are located at a multiple of 4-byte (i.e., their address has two zeroes at its LS bits). As a result, when the fields in a structure are laid out, some gaps may need to be introduced to satisfy the alignment requirement. These distances remain fixed, and must remain fixed, according to the C language semantics. In particular, if you change the size of gaps or introduce additional padding where it is not required, this will break working programs. As a result, it is not possible to use relative address randomization to defeat buffer overflows from one field of a struct to another. Indeed, many of the techniques for complete memory error protection do not offer any protection against such field-to-field overflows. With DSR, it is possible to simply use a different mask for each field, thus ensuring that such overflows become unexploitable. (To do this, it must be the case that the alias analysis report that the two fields in question aren't involved in aliasing --- this will not necessarily hold in all programs.)

Aside: Pointguard was a mechanism that was proposed earlier to protect against pointer corruption attacks. It relied on xor'ing pointer values with a bit-m ask. The problem with the technique was that it did not consider aliasing and hence would break some legitimate programs.

Special precaution is required to deal with aliasing. Two pointers pointing to the same data should not have different representations: otherwise, different masks may be associated with the access to the same data just because the pointer variables are different. For instance, consider:

```
struct XX a  =  malloc (sizeof (struct XX));
bzero(a,sizeof(struct XX));
```

where bzero is the standard C-library function used to zero out a block of memory. It is declared as follows:

```
Void bzero(char *b, int n);
```

As the function bzero will write '0' to all its fields while the fields may have a different representation for '0'. To ensure that this does not happen, the DSR technique uses a source code analysis to identify aliases, and ensures that the mask associated with a memory location stays the same, regardless of how this memory location is accessed.

Implementation:

Use source to source transformation. To achieve better performance, the technique is applied to buffers and pointers only. To minimize the likelihood of attacks due to the aliasing limitation mentioned above, it modifies the memory layout so that adjacent buffers use a different mask.

AAR doesn't cause any serious interoperability issues, RAR causes some (or at least suffers the problem of needing source code access.) But DSR introduces interoperability issues with existing libraries. To make it work correctly, all the libraries have to be analyzed and transformed. (To a varying degree, this is a common problem with all the memory protection mechanisms.)