

Effectiveness of Address Space Randomization

CSE 509 Notes by Ganesh Sangle.

Buffer Overflow Attacks

These attacks are possible because attacker has an idea of the layout of the memory. If he does not know the exact layout, he would try to use techniques like landing pads etc. to cause a buffer overflow attack.

Since the attack is based on this knowledge, if we take away this knowledge from the attacker, we can create a good defense against the buffer overflow attacks.

Memory Layout :

Kernel Memory (no read/write for user process)
Stack
Mmap/ libraries
Heap
BSS + Data segment
.txt

As seen in the above memory layout, to randomize the addresses, we can change the **start** addresses of:

- Stack
- Heap
- Mmap/libs

Changing start address of Stack : This is easy because all data access within the stack is relative to the start of the stack address.

However, if we just change the start of the stack, the attacker can still get around this defense by using trampoline attack, which does not depend on any guessing addresses on the stack. In trampoline attack, the attacker puts in the return address the address of jump instruction to some register content. Thus, changing the start address of stack is not much effective.

So, we now also shift the text/Mmap and the heap address. Changing the Mmap area is not difficult since it holds the libraries, which are anyways meant to be moved around easily. Similarly, moving text area is also not much difficult.

Caveats: Mmap and text area have to be aligned on page boundaries (of 4K size), which means that the least significant 12 bits are zero (4K needs 12 bits to be represented). Thus, of the 32 bits of address, for text/Mmap area, only 20 bits of randomization is possible.

Also, Mmap area needs to be somewhere between the stack and heap memory area, thereby limiting the addresses that it can take, thus removing 2-4 bits of more address bits.

Thus, we are left with just 16 bits of randomization of text/mmap area.

Similar arguments can be applied for **heap**, which **also provides for around 20 bits of randomization.**

This is not enough, given the computational power of the attacker.

Next point to note is, we are just changing the start addresses, but internal contents of these areas remain unchanged. Once the start of the segment is known, any internal area can be obtained using same offsets that is present on other machines, since there is no randomization internally.

Example of how to exploit this: Search the memory to find the position of sleep system call. Since the offset of sleep is same on attacker's machine too, thus he can find the offset of sleep system call from the start of segment on his machine, thus he gets the offset on victims machine too. Once he finds the start address, he can find the location of system call using offset determined from his machine and then effect a return to libc attack.

Thus attack can be described as follows:

For i in 0 to 2^{16} randomly chosen

Guess the address of sleep using i and try to invoke sleep using buffer overflow.

If sleep was successfully (in which case there will be a delay of 16 sec)

- a. Find the address of start of segment
 - b. Use this to find the address of system call and invoke the system call using buffer overflow attack.
-

This attack took about 200 sec which is approximately 4 minutes.

Thus, we can say that 16 bits can give us 4 minutes of security.

Thus, we see that even if we provide randomization for start addresses of stack, heap and text segment addresses, the attacker can just break one segments randomization using brute force, and all other randomizations prove useless.

So, what are the suggestions by the authors of this paper to prevent such attacks:

1. Use 64 bit systems

With 64 bit systems, mmap gives about 40 bits of security, which is equivalent to 2^{24} x 4 minutes or approximately 100 years of security. This is long enough for the victim to detect that an attack is underway. (One might try to use a botnet to attack one machine, reducing 100 years to some lesser value, but still if a botnet of 10 million machines when tries to attack one single machine, it can be detected due to increased and varied network traffic.)

So why is Address Space Randomization (ASR) popular even if it is not strong ?

ASR does not require changes to source code, may require some recompilations on programmers part. Since programmer has little work to do, and all of it is done by OS, it can be deployed easily. Moreover, compatibility with other libs is also good. Due to these reasons it seems to be quite a favorite.

Note: The effectiveness of address space randomization depends on how well we can 'randomize' the memory – ie. how good a random number generator we have. But now a days, most OS's come with good RNG's so we need not worry about this issue.

2. Randomizing internal contents of Stack/Heap/Mmap

Lets see on what all can be randomized in each segment and how much effort will it take :

a. Stack

- Local Variable ordering can be randomized
- Interframe random padding can be added
- Frames can be reordered (far fetched) on a stack
- Formal parameters can be randomly ordered (this is a bit difficult because if we have code from two different compilers, they should agree between themselves the order of parameter passing)

b. Code Segment

- Function ordering
 - The compiler can choose random instructions for performing an operation from allowed set of instructions for that operation. Example, to store zero in a register, the compiler can generate following piece of code : xor any number with itself, 'and' with zero the register contents, or move zero into register.
 - The compiler can also randomize basic blocks in a function and use goto jumps to create proper control flow.
- Random padding can be added
- Polymorphic code can be used : a code segment can be encrypted and it also has its decryption logic stored with itself.

C. Heap

- Inter allocation padding

- Random memory allocation (use of random fit method rather than best fit or other methods)
- Can data structures be randomized anyhow ? Not easily, because languages like C define the layout of data structure's like 'struct'.

In all above cases, we produce randomization within a single page content, hence we get around 2^{12} bits of randomization. Some of the above ideas are part of compile time randomizations. However, compile time randomization are ineffective, if the attacker has access to local machine, where he can compile stuff and find out the layout.