CS3500 Operating Systems
Course Project: Address Space Layout Randomisation

Submitted by:
CS18B025 M Harini Saraswathy
CS18B045 Rishika Varma K

December 5, 2020

Introduction:

Address Space Layout Randomisation (ASLR) is a security technique in which the address at which executable, stack, heap, libraries, etc are loaded at randomised locations as opposed to being loaded at the same fixed place. This technique helps in preventing buffer overflow attacks as the address at which each part of code and data is unknown and changes each time it is run.

ASLR increases the control-flow integrity of a system by making it more difficult for an attacker to execute a successful buffer-overflow attack by randomizing the offsets it uses in memory layouts. ASLR is used today on Linux, Windows, and MacOS systems. In this project, we implemented ASLR in the xv6 operating system.

Advantages and Disadvantages of ASLR:

Advantages:

ASLR (Address Space Layout Randomization) is a memory exploitation mitigation technique used on both Linux and Windows systems. Address Space Layout Randomization (ASLR) is a memory-protection process for operating systems that guards against buffer-overflow attacks. It helps to ensure that the memory addresses associated with running processes on systems are not predictable, thus flaws or vulnerabilities associated with these processes will be more difficult to exploit.

Disadvantages:

The effectiveness of ASLR is dependent on the entirety of the address space layout remaining unknown to the attacker. In addition, only executables that are compiled as Position Independent Executable (PIE) programs will be able to claim the maximum protection from ASLR technique because all sections of the code will be loaded at random locations.

Modifications required to incorporate ASLR into xv6

Randomisation function

Approach:

The first step is implementing the randomise function that is required for generating random numbers which are used as the offsets to the addresses at which stack, executable, etc are to be loaded. This was implemented using the Lehmer random number generator algorithm (Source: Wikipedia). The algorithm which was implemented is shown below.

```
uint32_t lcg_parkmiller(uint32_t *state)
{
    // Precomputed parameters for Schrage's method
    const uint32_t M = 0x7fffffff;
    const uint32_t A = 48271;
    const uint32_t Q = M / A;    // 44488
    const uint32_t R = M % A;    //  3399

    uint32_t div = *state / Q;  // max: M / Q = A = 48,271
    uint32_t rem = *state % Q;  // max: Q - 1     = 44,487

    int32_t s = rem * A;    // max: 44,487 * 48,271 = 2,147,431,977 = 0x7fff3629
    int32_t t = div * R;    // max: 48,271 *  3,399 =   164,073,129
    int32_t result = s - t;

    if (result < 0)
        result += M;

    return *state = result;
}
```

Implementation:

We implemented the above function in the file sysfile.c and the initial value of the seed used was the value of ticks. We have been kept track of this value in a variable hashticks using the ticks value in trap.c.

Code:

```
static unsigned random_seed = 1;
static int present = 0;

unsigned lcg_parkmiller()
{
    const unsigned N = 0x7fffffff;
    const unsigned G = 48271u;
    unsigned div = random_seed / (N / G);
    unsigned rem = random_seed % (N / G);

    unsigned a = rem * G;
    unsigned b = div * (N % G);

    return random_seed = (a > b) ? (a - b) : (a + (N - b));
}

uint64 random(int start, int end) {
    if(present == 1){
        return (lcg_parkmiller() % (end-start))+start;
    }
    random_seed = hashticks;
    present = 1;
    return (lcg_parkmiller() % (end-start))+start;
}
```

Adding support to enable aslr for a particular process

Approach:

This was achieved by adding a syscall that enables a variable signifying usage of aslr and an aslr.c file which can be called from user space. For a process to be run with aslr then we must call the aslr process with 2 arguments, the first being the name of the process to be run with aslr and the second argument is the argv for that process.

Implementation:

We added an aslr_var field in the proc structure to signify whether process is run with aslr. We added a syscall called aslr which sets the aslr_var field in proc which was initially 0 to 1. The aslr_var field in the proc structure is initialised to 0 in allocproc() (the constructor) and freeproc() (the destructor). We created a file aslr.c that calls the aslr syscall which in turn enables the aslr_var field in the proc for that process. After it returns from the syscall, exec is called on the process to be run whose name is given as the first argument to the aslr process and the remaining part of argv apart from the first string as the argv for the process (second argument in exec) . Thus aslr can be enabled for a particular process by calling aslr process with arguments as the name of the process to be run and its arguments.

Code:

```
uint64
sys_aslr(void)
{
  myproc()->aslr_var=1;
  return 0;
}

```

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
  aslr();
  exec(argv[1], (argv+1));
  exit(0);
}
```

Incorporating loading of executable at random offset

Approach:

We generate a random offset using the randomize function and start loading the executable from that address. As it is randomly generated it may not be page aligned and so the implementation of loadseg must be appropriately changed to take care of this issue.

Implementation:

A variable load_offset is maintained which is 0 if en_aslr is disabled and a random number between 0 to 1000 left shifted by 4 if it is enabled. This is the offset at which the executable and relocation addresses are shifted by. The loadseg function is then called with this offset added to the virtual address. In the loadseg function, the starting part of the segment which is not page aligned is loaded separately according to the size in a way similar to how the ending part of the segment that is not page aligned is loaded, and the rest of the segment is loaded through the for loop as was the case earlier (no modification in that aspect).

Code:

```c
if(en_aslr)
  load_offset = random(0, 1000) << 4;
else load_offset = 0;
//load_offset = PGROUNDUP(load_offset);
printf("load offset: 0x%x\n", load_offset);

// Read in ELF_PROG_LOAD programs into memory
sz = uvmalloc(pagetable, 0, load_offset);
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, 0, (uint64)&ph, off, sizeof(ph)) != sizeof(ph)) {
    printf("exec: readi error\n");
    goto bad;
  }
  if(ph.type != ELF_PROG_LOAD)
    continue;
  if(ph.memsz < ph.filesz) {
    printf("exec: memsz smaller than filesz error\n");
    goto bad;
  }
  if((sz = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz + load_offset)) == 0) {
    printf("exec: uvmalloc error\n");
    goto bad;
  }
  if(loadseg(pagetable, ph.vaddr + load_offset, ip, ph.off, ph.filesz) < 0) {
    printf("exec: loadseg error\n");
    goto bad;
  }
}
```

```c
// Load a program segment into pagetable at virtual address va.
// and the pages from va to va+sz must already be mapped.
// Returns 0 on success, -1 on failure.
static int
loadseg(pagetable_t pagetable, uint64 va, struct inode *ip, uint offset, uint sz)
{
  uint i, n;
  uint64 pa;

  // get the positive page offset of the va from the beginning of the page
  uint64 first_va = PGROUNDDOWN(va);
  uint64 pg_offset = va - first_va;

  // manually fill the first page, which might not be page aligned
  pa = walkaddr(pagetable, first_va);
  if (pa == 0)
    panic("loadseg: address should exist");
  // fill page with zeroes
  memset((void*)pa, 0, PGSIZE);
  // fill rest of page
  n = (sz < PGSIZE - pg_offset)? sz : PGSIZE - pg_offset;
  if(readi(ip, 0, (uint64)pa + pg_offset, offset, n) != n)
    return -1;
  offset += n;
  sz -= n;

  // use for loop for remaining pages
  for(i = PGSIZE; sz > 0; i += PGSIZE){
    pa = walkaddr(pagetable, first_va + i);
    if(pa == 0)
      panic("loadseg: address should exist");
    // zero the page
    memset((void*)pa, 0, PGSIZE);
    // fill the page or until there are no bytes left to write
    n = (sz < PGSIZE)? sz : PGSIZE;
    if(readi(ip, 0, (uint64)pa, offset, n) != n)
      return -1;
    offset += n;
    sz -= n;
  }

  return 0;
}
```

Incorporation of changing corresponding relocation offsets

Approach:

The relocation section header consists of the offsets and other information at which relocatable attributes are loaded. As the executable has been loaded at an offset the corresponding relocation offsets also need to be modified. This is done by going through all the section headers till we reach the relocation section header and then we increase the offset of the corresponding entries to be changed by load_offset value.

Implementation:

We created 2 extra structures in elf.h corresponding to section header and relocation entries respectively. These structures were used to read each section header from the elfhdr struct to check which of them is the relocations section header and to read each relocation entry from the section header from which the type of entry is read and the offset is changed (increased by load_offset) correspondingly if required.

Code:

```c
// Get Section Headers
for(i=0, off = elf.shoff; i < elf.shnum; i++, off += elf.shentsize) {
  if (readi(ip, 0, (uint64)&sh, off, elf.shentsize) != elf.shentsize)
    goto bad;

  int nr = (sh.type ^ 4);
  if (!nr) {
    // found section header for relocations
    // read through each relocation
    for (int sectoff = 0, relocnum = 1; sectoff < sh.size; sectoff += sh.entsize, relocnum++) {
      int size = readi(ip, 0, (uint64)&reloc, sh.offset + sectoff, sh.entsize);

      if((reloc.info&0xffffffffL) == 2 || (reloc.info&0xffffffffL) == 3 || (reloc.info&0xffffffffL) == 5){
        if (copyin(pagetable, (char*)&instr, (uint64)reloc.offset + load_offset, 8) != 0)
          goto bad;
        instr += load_offset;
        if (copyout(pagetable, (uint64)reloc.offset + load_offset, (char*)&instr, 8) != 0)
          goto bad;
      }

      if (size != sizeof(struct elfrel))
        goto bad;
    }
  }
}
```

Adding support for stack offset

Approach:

The other aspect of aslr is initialising stack at a random address as well. A random number of pages are left empty at the start after which the stack page is made.

Implementation:

A variable stack_offset is created and initialised to 2 if aslr is disabled or to a random number between 2 and 1000 obtained using randomize function if aslr is enabled. A total memory of the required size of the process that has been kept track of (rounded up to page align) and stack_offset number of pages is allocated. sp is then initialised to the size of memory that has been initialised (which is effectively the address just after the end of the stack page) and stackbase as the address of the start of the stack page (sp - page size). The rest of the exec process remains same as earlier in regards to creating new page directory, trapframe, etc.

Code:

```c
// Allocate random number of pages at the next page boundary.
// Use the last one as the user stack.
if(en_aslr)
  stack_offset = random(2, 1000);
else
  stack_offset = 2;
sz = PGROUNDUP(sz);
if((sz = uvmalloc(pagetable, sz, sz + stack_offset*PGSIZE)) == 0)
  goto bad;
uvmclear(pagetable, sz-stack_offset*PGSIZE);
sp = sz;
stackbase = sp - PGSIZE;

// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
  if(argc >= MAXARG)
    goto bad;
  sp -= strlen(argv[argc]) + 1;
  sp -= sp % 16; // riscv sp must be 16-byte aligned
  if(sp < stackbase)
    goto bad;
  if(copyout(pagetable, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
    goto bad;
  ustack[argc] = sp;
}
ustack[argc] = 0;
```

Modifications made to show proof of working

To show that the technique is working we printed certain values in exec.c. The values that will change are the place at which executable is loaded, in which case the entry point at which the implementation enters (main function) changes for each time the process is run. Thus effectively the initial entry point address is stored in the epc register and so this is printed to check if it varies. Similarly, the other thing that has been randomized is the starting address of the stack which is the same as the initial value of sp and so to show the randomisation in sp this is also printed. An extra message is printed to indicate whether aslr is enabled or disabled.

Code:

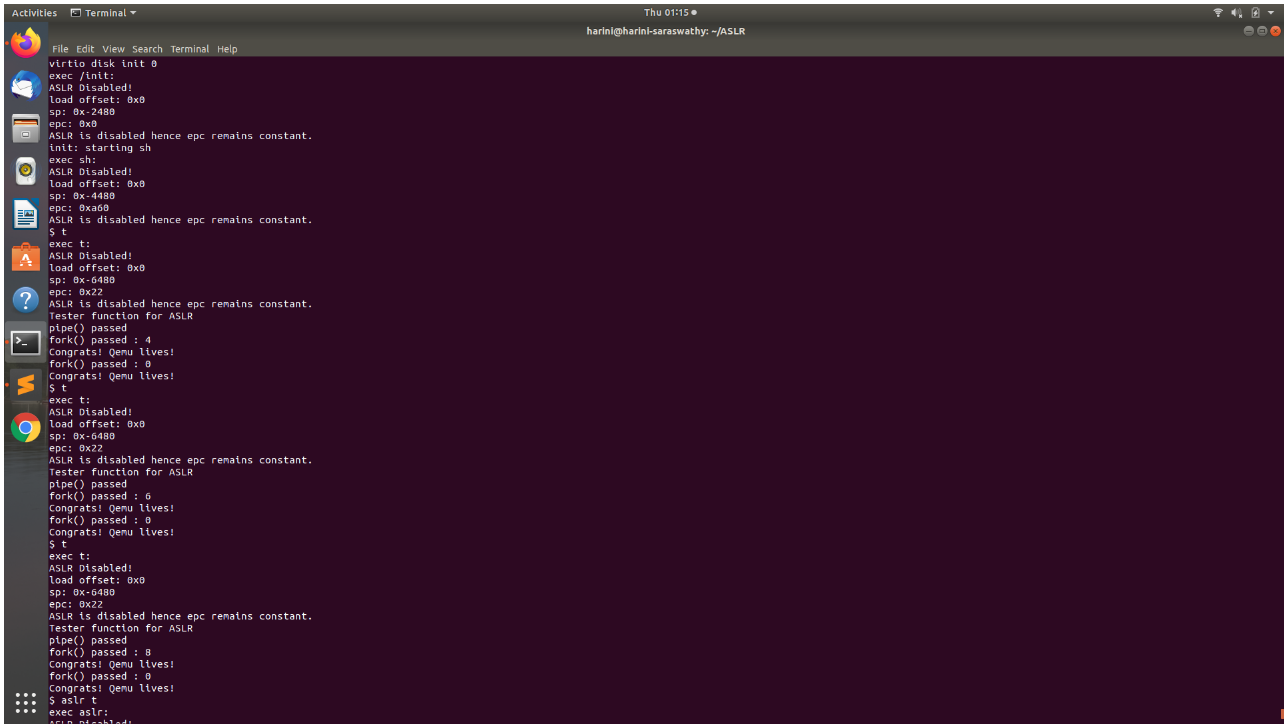
```c
  // Save program name for debugging.
  for(last=s=path; *s; s++)
    if(*s == '/')
      last = s+1;
  safestrcpy(p->name, last, sizeof(p->name));

  // Commit to the user image.
  oldpagetable = p->pagetable;
  p->pagetable = pagetable;
  p->sz = sz;
  p->tf->epc = elf.entry + load_offset;  // initial program counter = main
  p->tf->sp = sp; // initial stack pointer
  proc_freepagetable(oldpagetable, oldsz);

  printf("sp: 0x%x\n", r_sp());
  printf("epc: 0x%x\n", p->tf->epc);
  if(en_aslr)
    printf("ASLR is enabled hence epc changes with each run of aslr.\n");
  else
    printf("ASLR is disabled hence epc remains constant.\n");
  return argc; // this ends up in a0, the first argument to main(argc, argv)

 bad:
  if(pagetable)
    proc_freepagetable(pagetable, sz);
  if(ip){
    iunlockput(ip);
    end_op(ROOTDEV);
  }
  return -1;
}
```

Code of the process that is tested:

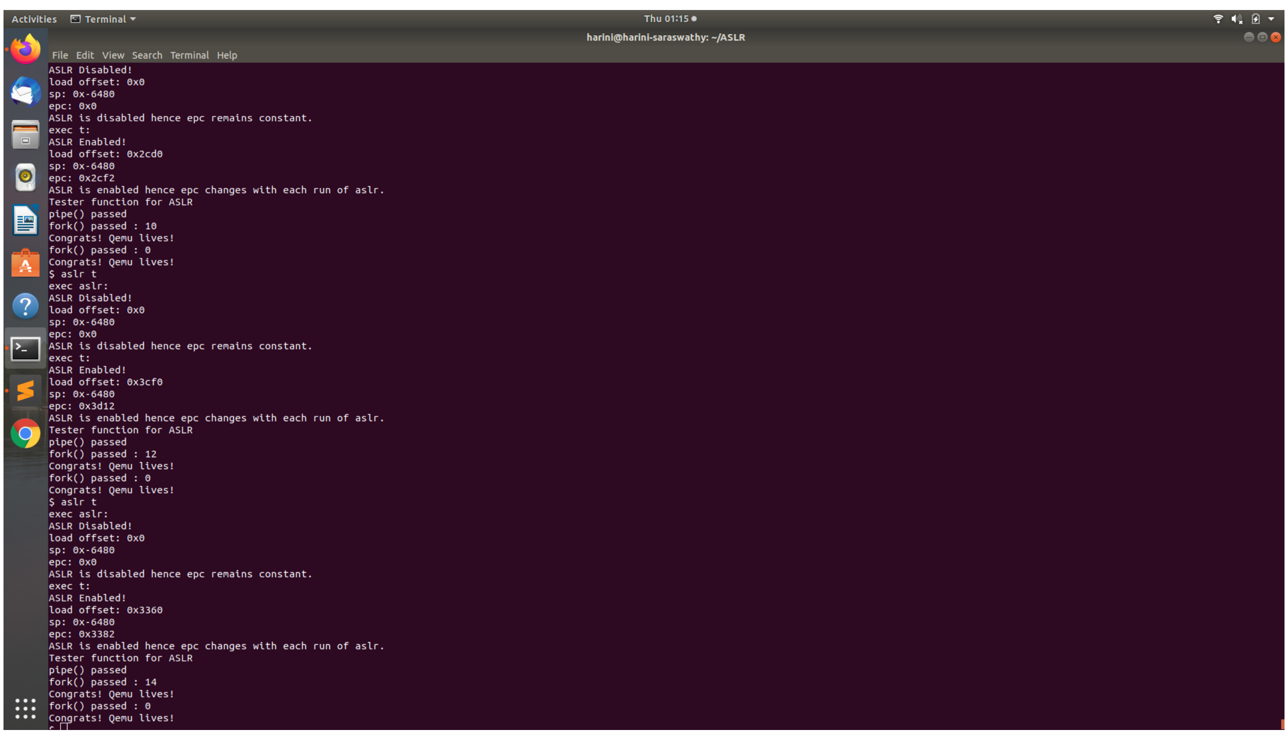
```c
#include "kernel/types.h"
#include "user/user.h"
#include "kernel/fcntl.h"

int temp() {
  printf("Tester function for ASLR\n");
  return 1;
}

int main()
{
  if(temp() != 1)
    exit(1);

  int fds[2];

    if(pipe(fds) != 0){
      printf("pipe() failed\n");
      exit(1);
    }
    else printf("pipe() passed\n");

    int pid = fork();
    printf("fork() passed : %d\n", pid);

    printf("Congrats! Qemu lives! \n");
    exit(0);
}
```

Output:

Aslr disabled:



Aslr enabled:



References:

- https://nptel.ac.in/courses/106/106/106106199/
- https://en.wikipedia.org/wiki/Lehmer_random_number_generator
- https://www.reddit.com/r/osdev/comments/8b4tnj/aslr_implementation/
- https://iitd-plos.github.io/os/2020/ref/os-arpaci-dessau-book.pdf
- http://pages.cs.wisc.edu/r̃emzi/OSTEP/
- https://github.com/johnmwu/xv6-aslr
- https://github.com/tenkjm/xv6-aslr
- https://github.com/typingkla/xv6-aslr
- https://www.networkworld.com/article/3331199/what-does-aslr-do-for-linux.html
- https://tc.gts3.org/cs3210/2016/spring/p/prop-slides.pdf
- https://blog.morphisec.com/aslr-what-it-is-and-what-it-isnt/
- https://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-54839.html
- https://refspecs.linuxbase.org/elf/gabi4+/ch4.reloc.html
- https://stackoverflow.com/questions/16847741/processing-elf-relocations-understanding-the-relocs-symbols-section-data-an
- http://blog.k3170makan.com/2018/10/introduction-to-elf-format-part-vi_18.html