COL 331: OS ASSIGNMENT 3

Assignment 3: Report

2023-04-30

Reedam Dhake — 2020CS10372
Nikhil UNavekar — 2020CS10363

Contents

1 Changes Made

- Added 3 function `read_aslr_flag` ,`set_randoffset_text` and `set_randoffset_stack` in `exec.c`
- Added 2 functions `get_pa` and `new_virtual_memory_loader` in `vm.c`
- Defined 3 files - `alsr_flag` (for ALSR value), `gen_expliot.py` and `random.h`. `gen_expliot.py` generates payload and `random.h` is the random integer generator.

2 Address Space Layout Randomization

CHANGES IN VM.C

```
uint get_pa(pde_t *pte, pde_t *pgdir, char *first_va, int alloc){
  if((pte = walkpgdir(pgdir, (char * )first_va, 0)) == 0)
    panic("loaduvm: address should exist");
  else
    return PTE_ADDR(*pte);
}

int new_virtual_memory_loader(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz){
  uint i, pa, n,va=(uint)addr;
  pte_t *pte;
  uint first_va = PGROUNDDOWN(va);
  uint pg_offset = va - first_va;
  pa = get_pa(pte, pgdir, (char*)first_va, 0);
  n = (sz < PGSIZE - pg_offset) ? sz : PGSIZE - pg_offset;
  if(readi(ip, P2V(pa + pg_offset), offset, n) != n)
    return -1;
  offset += n;
  sz -= n;
  for(i = PGSIZE; sz > 0; i += PGSIZE){
    pa = get_pa(pte, pgdir, (char*)(first_va + i), 0);
    n = (sz < PGSIZE) ? sz : PGSIZE;
    if(readi(ip, P2V(pa), offset, n) != n)
      return -1;
    offset += n;
    sz -= n;
  }
  return 0;
}
```

- The get_pa function is called by the new_virtual_memory_loader function to retrieve the physical address corresponding to a given virtual address.It takes as input a page table entry pointer pte, a pointer to the first byte of the page containing the virtual address first_va, a page directory entry pointer pgdir, and an integer alloc for new page allocation.
- It first calls the walkpgdir function to obtain a pointer to the page table entry corresponding to the virtual address first_va in the page directory pgdir. If walkpgdir returns a null pointer, indicating that the page table entry is not present,error is returned. Elsee, the function returns the physical address corresponding to the page table entry by making the page table entry with the PTE_ADDR , which extracts the physical address from the page table.

- In the new_virtual_memory_loader function, the thiss function is used to retrieve the physical address of the page containing the first word of the virtual address addr, followed by retrieving the physical addresses of each subsequent page that needs to be loaded with data from the file.

  new_virtual_memory_loader takes a page directory entry pointer pgdir, a virtual address addr, an inode ip, an offset offset, and a size sz as inputs.

- This function loads sz bytes of data from the file referred to by the inode ip at the specified offset into the virtual memory at the address addr, using the page directory pgdir.

- It first caculates the page offset pg_offset of the virtual address addr from the beginning of the page, and then retrieves the physical address pa corresponding to the page containing the first byte of addr using the get_pa function as described above.

- It then reads the first n bytes of data from the file at the specified offset into the physical memory starting at P2V(pa + pg_offset), where P2V is usedd to convert the a physical address to a virtual address. It then subtracts n from sz to indicate that n bytes have been loaded.

- FInally it loops to load the remaining bytes. In each iteration, the function calculates the physical address pa of the next page of virtual memory by adding the page size PGSIZE to the previous physical address pa. Then it reads the next n bytes of data from the file at the specified offset into the physical memory starting at P2V(pa). The function then subtracts n from sz to indicate that n more bytes have been loaded.The loop continues until all sz bytes have been loaded. Error returns -1 else return 0.

CHANGES IN EXEC.C

```
int read_aslr_flag(struct inode *ip){
  char temp[1]={0};
  ip = namei("aslr_flag");
  ilock(ip);
  readi(ip,&temp[0],0,8);
  iunlockput(ip);
  if(temp[0]=='1'){
    return 1;
  }
  else{
    return 0;
  }
}

int set_randoffset_text(int asfl){
  int offset1=random_num_gen(1,4000);
  offset1*=32;
  if(asfl==0){
    return 0;
  }
  return offset1;
}

int set_randoffset_stack(int asfl){
  int extra_pgs=random_num_gen(2,30);
  int offset2=2+extra_pgs;
  if(asfl==0){
    offset2=2;
```

```
  }
  return offset2;
}
```

```
int
exec(char *path, char **argv)
{
   ...
  begin_op();
  int asfl=read_aslr_flag(ip);
  ....
  // Check ELF header
  int offset1=set_randoffset_text(asfl);
  sz = 0;
  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
      goto bad;
    if(ph.type != ELF_PROG_LOAD)
      continue;
    if(ph.memsz < ph.filesz)
      goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
      goto bad;
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz+offset1)) == 0)
      goto bad;
    if(ph.vaddr % PGSIZE != 0)
      goto bad;
    if(new_virtual_memory_loader(pgdir, (char*)(ph.vaddr+offset1), ip, ph.off, ph.filesz) < 0)
    goto bad;
  }
  iunlockput(ip);
  end_op();
  ip = 0;
  int offset2=set_randoffset_stack(asfl);
  sz = PGROUNDUP(sz);
  if((sz = allocuvm(pgdir, sz, sz + offset2*PGSIZE)) == 0)
    goto bad;
  clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
  sp = sz;
  ....
}
```

- Start defines the 3 functions which are explained below and which we will be using the core exec.c function.
- The read_aslr_flag reads the value of a file named ”aslr_flag” and the falg value, 1 indicates that Address Space Layout Randomization (ASLR) is enabled, and 0 means its disabled. It takes a pointer to an inode structure as an argument, which is used to access the file.
- The set_randoffset_text generatess a random offset for the program text section based on the value of asfl. If asfl is 0, it returns 0, otherwise, it generates a random number between 1 and 4000, multiplies it by 32, and returnns it.
- The set_randoffset_stack generates a random offset for the stack section based on the value of flag asfl. If asfl is 0, it returns 2, otherwise, it generates a random number between 2 and 30 and adds 2 to it before returning the result. The exec function loads and executes a new program by replacing the

current process's memory image with the program thta is to be executed. This function takes the path of the executable and its arguments as parameters.

- First, it reads the value of the ASLR flag by calling the read_aslr_flag function, which opens and reads a file named "aslr_flag" to check if the ASLR feature is enabled or not. If ASLR is disabled, the offset valuees for text and stack segments will be set to their defoult values.
- After obtaining the ASLR flag, exec opens the executable file by calling namei function and checks the ELF header of the cprogram that is to run. If the ELF header is invalid, the funcction returns an error. Next it sets up a new kernel page table by calling setupkvm function. he old page directory is saved to oldpgdir to be restored later.
- Then, exec calculates a random offset value for the text segment by calling the set_randoffset_text function, which generates a random number between 1 and 4000 and multiplies it by 32. If ASLR is disabled, offset will be set to 0.
- If an attacker attempts to launch a buffer overflow attack against a program with ASLR enabled, they will not be able to predict the address where the malicious code will be loaded, as the address is randomized every time this program is executed.

  The random integer generator is described below -

RANDOM.H

```
int rand1(int rr){
    int rand = rr;
    rand=(rand+(ticks))%(85663);
    rand *= 12;
    rand += 8965;
    rand ^= 1463588;
    return rand;
}

int random_num_gen(int mini,int maxi){
    int rand = rand1(2);
    for(int i=0;i<ticks*5;i++){
        rand=rand1(rand);
    }
    rand = rand % 1000000009;
    rand=rand%(maxi-mini+1);
    cprintf("%d\n",rand);
    return mini+rand;
}
```

The rand1 function takes an integer rr as input and uses it as the seed for the random number generator. It generates a random numberr by performing a series of arithemetic operations on the seed value, including adding the current value of the ticks of system, adding a constant value, performing a bitwise XOR with another constant value,etc. The resulting value is returned as the random number. THis function is called by random_num_gen function . It It first calls rand1 with a seed value of 2, and then iteratively calls rand1 on the result of the previous call ticks*5 number of times to generate a more random value. It then takes the modulus of the resulting value with a large number. Finally, it takes the modulus of the range between the given minimum and maximum values and adds the minimum value to the result to generate a random number within the desired range of max and min.

3 Buffer Overflow Attack in XV6

```
import sys
f = open("payload", "wb")
f.write(b"aaaaaaaaaaaa" + b'a'*(int(sys.argv[1])) + b"\x00\x00" * 2)
f.close()
```

- THe above script can be used for to generate a payload for buffer overflow attack . As we know in a buffer overflow attack, an attacker tries to overwrite a buffer with more data than the buffer can hold, with the goal of overwriting adjacent memory locations. b”aaaaaaaaaaaa” string of 12 bytes (ASCII characters ’a’) serves as a buffer overflow trigger.
- The length of this string is the minimum amount of data required to overflow the buffer.
- THen it creates a string of variable length, depending on the command line argument passed to the script. This string is the payload that will overflow the buffer. It consists of ’a’ characters repeated input times. sys.argv[1] is the first argument. b”_x00_x00” * 2 aadds two null bytes to the end of the payload.
- Null bytes can be used to terminate strings, so adding them can help ensure that the payload does not corrupt other data beyond the intended buffer. An attacker could generate a payload of arbitrary length, and then pass the length as a command line argument when running the actual script.
- The resulting payload file could then be used as input to a vulnerable program that is to recieve a buffer overflow attack. When the program reads in data from the payload file, it will eventually hit the b”aaaaaaaaaaaa” string, which will trigger the overflow and allow the payload to overwrite adjacent memory locations.