CS380L: Lab 3

Hochan Lee

April 2019

1 Hardware Environment

The host machine consists of a single socket system with 64bit Intel i5-2467M CPUs with 2 cores and 4 hyper-threads. 8GB DRAM and 3072K last level cache (LLC) are used and Transparent Huget Pages (THP) is disabled. In addition, 4-way set associative and 32 entries for 2MB or 4MB pages, and 4-way set associative and 64 entries for 4KB pages are used for DTLB. Instruction TLB consists of 4KB pages, 4 way set associative, and 64 entries. L2 TLB has 1MB, 4-way set associative, and 64 byte line size. Shared 2nd-level TLB consists of 4KB pages, 4-way set associative, and 512 entries.

2 Software Environment

Operating system is Ubuntu 14.04.6 LTS and kernel version is Linux 4.4.0-142-generic. GCC version is 4.8.4 and GNU Binutils version 2.24 is used.

3 execv() in the Linux kernel

Operating system loads and executes the program when programs call execv(). Then, execve(), do_execve() and do_execve_common() (on the recent version, it changed to do_execveat_common()) are called in sequence. Below is pseudo-code for creating the memory image of a new process.

do_execveat_common()

1. File name check.
2. RLIMIT_NPROC check (RLIMIT_NPROC is max number of processes).
3. File open and create data structrue named bprm that maintains program information to be read.
4. Other memory related data structures are constructed (mm).
5. Fill the bprm by using the inode; set and copy argument information, and set uid/euid/guid/egid.
6. Copy file name, environment and argument vectors from old stack.
7. Call exec_binprm() which calls load_elf_binary() if the target program to be loaded is a ELF based file.

load_elf_binary()

1. Check simple consistency by checking e_ident and e_type fields of file header.
2. Load ELF program headers information.
3. Iterate all the program headers.

4. Check PT_INTERP segment in order to check whether interpreter is specified or not. Interpreter is used for shared libraries. If interpreter exists, then kernel loads it.
5. Check p_type of the program header whether it is PT_LOAD or not.
6. If the current header is about loadable segments, then kernel maps them into memory.
7. In case of BSS section, this area is basically set by 0.
8. Call create_elf_tables()
9. After then, call start_thread with e_entry and other information in order to transfer control to the new program.

create_elf_tables()

1. Create ELF interpreter information.
2. Create a stack with random padding.
3. Set the auxiliary vectors, argv pointers and envp pointers to corresponding strings.

4 Jump to entry point vs main function

To correctly load a program on the loader, we should jump to entry point, not main function location. The entry point points to _start and in this part, __libc_start_main is called. The __libc_start_main initializes the necessary execution environment: security checks, threading subsystem initialization, releasing dynamic shared object and registering the fini handler. After these initialization, __libc_start_main calls main() with proper arguments. If we skip these process and directly call the main(), then the necessary information is not initialized and the loaded program would not work correctly and be crashed.

5 Makefile: static linking and exploit disjoint memory regions

Figure 1: **Makefile**

```
default: clean apager dpager hpager test_input
    #gcc -static hloader.c -o hloader
apager:
    gcc -g -fPIE -fPIC -Bstatic -static -static-libgcc -Wl,-Ttext-segment=0x20000000 apager.c -o apager

dpager:
    gcc -g -Bstatic -fPIE -fPIC -Bstatic -static -static-libgcc -Wl,-Ttext-segment=0x20000000 dpager.c -o dpager

hpager:
    gcc -g -Bstatic -fPIE -fPIC -Bstatic -static -static-libgcc -Wl,-Ttext-segment=0x20000000 hpager.c -o hpager

test_input:
    gcc -g -Bstatic -fPIE -fPIC -Bstatic -static -static-libgcc -Wl,-Ttext-segment=0x4000000 helloworld.c -o helloworld
    gcc -g -Bstatic -fPIE -fPIC -Bstatic -static -static-libgcc -Wl,-Ttext-segment=0x4000000 zero.c -o zero
    gcc -g -Bstatic -fPIE -fPIC -Bstatic -static -static-libgcc -Wl,-Ttext-segment=0x4000000 test_malloc.c -o test_malloc
    gcc -g -Bstatic -fPIE -fPIC -Bstatic -static -static-libgcc -Wl,-Ttext-segment=0x4000000 test_rand.c -o test_rand
    gcc -g -Bstatic -fPIE -fPIC -Bstatic -static -static-libgcc -Wl,-Ttext-segment=0x4000000 test_bss.c -o test_bss

clean:
    rm -rf apager hpager dpager helloworld zero test_malloc

```

My loader and the test programs are linked statically with '-fPIE' (disable address space layout randomization), '-fPIC' (disable memory position independent code option), '-Bstatic', '-static' and '-static-libgcc' options. In addition, by fixing text segment, the loader and the test programs are located in disjoint address regions.

6 Implementation

6.1 All-at-once loader

All-at-once version maps all the program including BSS with anonymous mapping. All-at-once loader follows below flows. **All mmaped memroy is aligned based on page size. In addition, all mapped address is treated as absolute address by using MAP_FIXED option.**

1. Read ELF based binary file and map file header by using Elf64_Ehdr.
2. Elf64_Ehdr data structure contains program header data offsets on the file and the number of program headers on e_phoff and e_phnum fields respectively. Map the program headers onto Elf64_Phdr and traverse each program header.
3. Check whether p_type of the current program header is PT_LOAD or not. It it is, the corresponding segment should be mapped on the memory. Otherwise, skip below codes and traverse other headers.
4. Map all the segments onto memory. Each segment starts from **p_vaddr & ~(PAGE_SIZE-1)**. Size is **((p_filesz+(p_vaddr & (PAGE_SIZE-1)))+(PAGE_SIZE-1)) & ~(PAGE_SIZE-1)**. File offset is **p_offset-(p_vaddr & (PAGE_SIZE-1))**.
5. If the p_memsz is higher than p_filesz, it is BSS area. With additional MAP_ANONYMOUS flag, I also map the BSS segment onto the zero-filled memory.
6. After loading the program, construct new stack for the loaded program. In this case, I reused the loader program's stack. Calculate the current stack's size by traversing the memory and mmap() by that size. In that case, I used MAP_GROWSDOWN flag in order to make the mapped memory increase in downwards.
7. After mapping, modify argc and some auxiliary vector data such as AT_ENTRY, AT_PHNUM, AT_PHENT, and AT_EXECFN. Therefore, **I constructed the stack that is exactly same a stack constructed by the kernel.**
8. Lastly, clear registers including rdx, and jump to the entry point.

6.2 Demand loader

Demand loader maps just the first page. Then, it initializes signal sets (sigemptyset()) and sets new customized segmentation signal handler named seghandler. This handler is called whenever the loaded program accesses to unmapped and causes segmentation fault. In that case, the handler maps one page that contains accessed address onto the memory by reading corresponding part on the file. Other functions are the same with All-at-once loader.

6.3 Hybrid loader

Hybrid loader maps segments while loading the program except BSS segment. When the program tries to access BSS, it invokes segmentation fault. As the above demand loader does, this signal is caught by customized handler and maps one page containing the accessed memory address.

Prediction Optimization

I applied three stage prediction algorithm. It is similar with branch prediction optimization. It uses prediction counter which is increased until 2 whenever prediction is correct. However, a prediction fails, it is decreased until 0. So there are 0, 1 and 2 stages. I exploited basic prediction; just predict next consecutive page. If the counter is 0, then it just maps requested page. In case that counter is 1, it maps additional next page onto the memory. Similarly, in case of 2, it maps two consecutive next pages.

6.4 malloc()

malloc() is implemented in glibc and it basically exploits system call in order to allocate new memory. Therefore, malloc is done by kernel and my loader does not mmap() for the malloc() request. Therefore, without intervening of the loader, it successfully allocates the new requested memory.

6.5 overlapped area processing

Figure 2: **Find entry point of the loader**

```
320     char** overlap = envp;
321     uint64_t loader_addr;
322     while ( *(overlap++) == NULL ) ;
323     Elf64_auxv_t* auxv_ptr = (Elf64_auxv_t *) overlap;
324     for (; auxv_ptr->a_type != AT_NULL; auxv_ptr++) {
325         switch (auxv_ptr->a_type) {
326             case AT_ENTRY:
327                 loader_addr = auxv_ptr->a_un.a_val;
328                 break;
329         }
330     }
331
```

To process a case which the loaded program memory area is overlapped with the loader's, The loader tracks its entry point like Figure 2. Then, after reading file header of the loaded program, the loader checks whether the entry point of the loaded program is overlapped or not. If it is, the loader gracefully exits its execution.

7 Evaluation

In this section, I compared apager, dpager, hpager (using prediction optimization), and hpager-not-predict (not using prediction optimization) by using four test programs. All the experiments are performed third time and results are average of them. Basically, they are designed to show better results for each implementation. Figure 3 shows results of execution time and normalized memory consumption for the loaders. Each subsection analyzes and understand the results.

Figure 3: **Results**

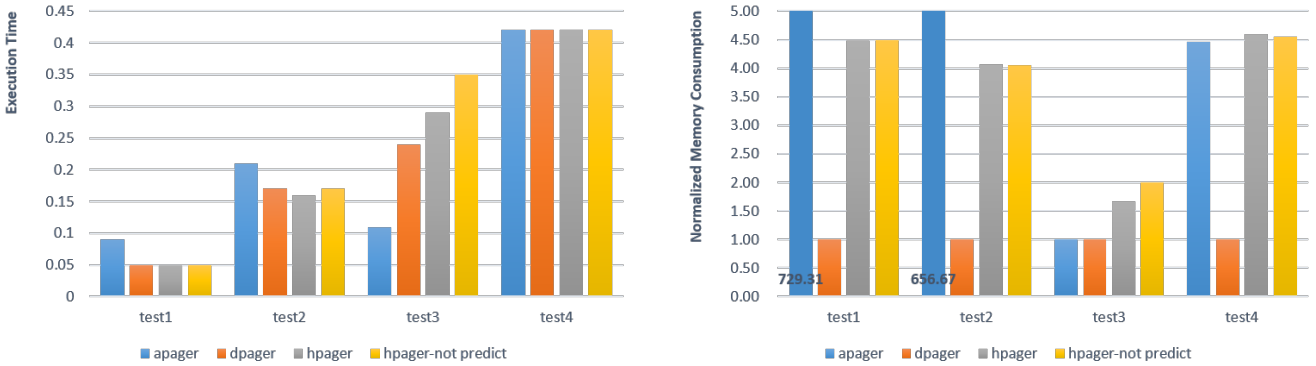

Figure 4: **Test1: Only access one item**

```c
#include <stdio.h>

#define BUCKET_SIZE 1<<25
int bucket[BUCKET_SIZE];

int main(void)
{
    int i;
    for (i = 0; i < BUCKET_SIZE; i++)
        bucket[0] = i;

    return 0;
}
```

7.1 Test program

7.1.1 Test1

Test1 program declares one 32MB integer array in the BSS segments. However, it does not touch items only except 0 indexed item. This application is designed to show the case that dpager shows the greatest memory efficiency compared to others. In that case, apager consumes extremely large memory compared to dpager since it allocates memory for all segments. Since hpager loaders also allocate data and text segments in the first phase, it shows higher memory consumption compared to dpager. On the contrary, dpager shows the best memory efficiency since it only allocates one page for each segment in initial phase. Moreover, the test1 requests memory for one item, i and j, the amount of demanded memory is also extremely low. In case of execution time, hpager and dpager show comparable results; **they show faster execution time than apager**. Since it accesses only one item, hpager's predict optimization is useless.

7.1.2 Test2

Figure 5: **Test2: Random access**

```c
#include <stdio.h>

#define BUCKET_SIZE 1<<25

int bucket[BUCKET_SIZE];
int i, j;

int main(void) {
    srand(time(NULL));

    for (i = 0; i < BUCKET_SIZE; i++)
        j = bucket[rand()%BUCKET_SIZE];

    return 0;
}
```

Test2 program declares one big integer array like the test 1 and randomly accesses it. In this case, dpager also shows the best memory efficiency due to the same reason of the above test. Random access cannot affect a lot to dpager because it anyway allocates memory whenever it is requested. In this experiment, hpagers' memory performance is slightly improved since this test approaches more items of the array than test1. However, prediction optimization does not effects a lot due to random access manner. apager shows the worst memory consumption since it basically allocates all memory for the array; even though some items are not accessed. **dpager and hpager show better execution time than apager.**

Figure 6: **Test3: Access all items**

```c
#include <stdio.h>

#define BUCKET_SIZE 1<<25
int bucket[BUCKET_SIZE];
int i,j;
int main(void)
{
    for (i = 0; i < BUCKET_SIZE; i++)
        j = bucket[i];

    return 0;
}
```

7.1.3 Test3

Test3 accesses all items on the array declared on BSS segment. apager shows great memory performance because it basically maps all memory including bss. Since apager does not need to suffer from segfault, execution time is the best. dpager also shows comparable memory consumption because it just maps necessary amount of memory. However, since dpager always suffers from entering kernel for every segfault, it takes longer time than apager; with the same reason, hpager also takes longer time than apaager. In this experiment, hpager prediction works well since test3 sequentially accesses item. Therefore, it is easy to predict the next page. **hpager with prediction optimization shows faster execution time than hpager without prediction optimization.**

7.1.4 Test4

Figure 7: **Test4: Malloc**

```c
#include <stdio.h>
#include <stdlib.h>

#define BUCKET_SIZE 1<<25
int i,j;
int main(void)
{
    char* test;
    for (i = 0; i < BUCKET_SIZE; i++) {
        test = (char *) malloc (100*sizeof(char));
        free(test);
    }

    return 0;
}
```

In this experiment, I expected that all cases use the same amount of memory since malloc does not affect on mmapped memory consumption. However, interestingly, dpager uses smaller amount of memory than others. In case of apager and hpager, it shows comparable memory consumption. In case of execution time, four implementation shows similar results.

7.1.5 Test5

Figure 8: **Test5: NULL return**

```c
#include <stdio.h>
int
main() {
   int *zero = NULL;
   return *zero;
}
```

For this code, all loader should invoke segmentation fault. In this code, apager and hpager invoke segmentation fault. In bpager implementation, I catch segfault and when this address access is invalid, then I ignore this request. Therefore, no memory is allocated for the incorrect address request, and however, the program keep asking memory for the address. Previously therefore, bpager enters infinity loop. **However, I modified it to invoke SIGSEGV again when invalid address is requested. Finally, it becomes double faults and user program would get segmentation faults.**

8 Additional Information

To test time and memory consumption comfortably, I made MACRO named 'DEBUG'. It is located on the top of the source codes and if it is set, then each loader will print out mmap() start address, offset and size. Otherwise, nothing will be printed and we can correctly test the execution time performance.

9 Spend Time

60 hours: due to compilation problem.