1-1

1-1. 어떤 함수를 관찰지점으로써 이용하고, 어떤 eBPF 자료구조 및 요소들을 활용하여 모니터링 하는지 확인

- execsnoop.py
  1. 관찰지점

     execve 시스템콜 호출될 때마다 syscall__execve()함수 호출

     execve 시스템콜 return할 때마다 do_ret_sys_execve()함수 호출

```
# initialize BPF
b = BPF(text=bpf_text)
execve_fnname = b.get_syscall_fnname("execve")
b.attach_kprobe(event=execve_fnname, fn_name="syscall__execve")
b.attach_kretprobe(event=execve_fnname, fn_name="do_ret_sys_execve")
```

  2. 모니터링에 활용되는 자료구조 및 요소

```
struct data_t {
    u32 pid;  // PID as in the userspace term (i.e. task->tgid in kernel)
    u32 ppid; // Parent PID as in the userspace term (i.e task->real_parent->tgid in kernel)
    u32 uid;
    char comm[TASK_COMM_LEN];
    enum event_type type;
    char argv[ARGSIZE];
    int retval;
};
```

  3. 고찰

     execve호출 통해서 새로운 프로세스의 실행을 추적하기 위해 사용된다. execve이 호출될 때마다 설정해놓은 함수가 호출되어 원하는 정보를 나타낸다. pid, ppid, 커맨드, 리턴값 등의 정보를 나타낸다.

- oomkill.py
  1. 관찰지점

     oom_kill_process()가 호출될때 이 함수를 실행시킨다.

```
void kprobe__oom_kill_process(struct pt_regs *ctx, struct oom_control *oc, const char *message)
{
```

2. 모니터링에 활용되는 자료구조 및 요소

```
struct data_t {
    u32 fpid;
    u32 tpid;
    u64 pages;
    char fcomm[TASK_COMM_LEN];
    char tcomm[TASK_COMM_LEN];
};
```

3. 고찰

   Out-of-Memory(OOM) kill 발생을 추적하기 위해 사용된다. oom_kill_process가 호출될 때마다 설정 해놓은 함수가 호출되어 원하는 정보를 나타낸다. pid, page정보, 명령어정보등의 정보를 나타낸다.

- filelife.py

1. 관찰지점

```
# initialize BPF
b = BPF(text=bpf_text)
b.attach_kprobe(event="vfs_create", fn_name="trace_create")
# newer kernels (say, 4.8) may don't fire vfs_create, so record (or overwrite)
# the timestamp in security_inode_create():
b.attach_kprobe(event="security_inode_create", fn_name="trace_create")
b.attach_kprobe(event="vfs_unlink", fn_name="trace_unlink")
```

2. 모니터링에 활용되는 자료구조 및 요소

```
struct data_t {
    u32 pid;
    u64 delta;
    char comm[TASK_COMM_LEN];
    char fname[DNAME_INLINE_LEN];
};
```

3. 고찰

   Lifetime이 짧은 File을 추적하기 위해 사용된다. vfs_create, security_inode_create, vfs_unlink가 호출될 때마다 설정해놓은 함수가 호출되어 원하는 정보를 나타낸다. 파일존재 시간, pid, 파일이름등을 나타낸다.

- biosnoop.py

1. 관찰지점

```
# initialize BPF
b = BPF(text=bpf_text)
b.attach_kprobe(event="blk_account_io_start", fn_name="trace_pid_start")
if BPF.get_kprobe_functions(b'blk_start_request'):
    b.attach_kprobe(event="blk_start_request", fn_name="trace_req_start")
b.attach_kprobe(event="blk_mq_start_request", fn_name="trace_req_start")
b.attach_kprobe(event="blk_account_io_done",
    fn_name="trace_req_completion")
```

2. 모니터링에 활용되는 자료구조 및 요소

```
// for saving the timestamp and __data_len of each request
struct start_req_t {
    u64 ts;
    u64 data_len;
};

struct val_t {
    u64 ts;
    u32 pid;
    char name[TASK_COMM_LEN];
};

struct data_t {
    u32 pid;
    u64 rwflag;
    u64 delta;
    u64 qdelta;
    u64 sector;
    u64 len;
    u64 ts;
    char disk_name[DISK_NAME_LEN];
    char name[TASK_COMM_LEN];
};
```

3. 고찰

Block I/O 요청의 발생을 추적하기 위해 사용된다. blk_account_io_start, blk_start_request, blk_mq_start_request, blk_account_io_done이 호출될 때마다 설정해놓은 함수를 호출하고 원하는 정보를 나타내준다. 출력으로는 시간, 섹터 정보, 요청 크기, read/write 타입, 디스크정보, pid등의 정보를 나타낸다.