sample binary를 strace했을 시에 나오는 처음 로직.

root@9eb26f7bbdb1:/shared/cpp# strace -fFi ./setmagic

```
[00007fafdff6b777] execve("./setmagic", ["./setmagic"], [/* 17 vars */]) = 0
[00007fde880974b9] brk(NULL)             = 0xc07000
[00007fde88098387] access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or
directory)
[00007fde8809847a] mmap(NULL, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fde882a0000
[00007fde88098387] access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or
directory)
[00007fde88098327] open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
[00007fde880982b4] fstat(3, {st_mode=S_IFREG|0644, st_size=58070, ...}) = 0
[00007fde8809847a] mmap(NULL, 58070, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fde88291000
[00007fde88098427] close(3)              = 0
[00007fde88098387] access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or
directory)
[00007fde88098327] open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
[00007fde88098347] read(3,
"\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"..., 832) = 832
[00007fde880982b4] fstat(3, {st_mode=S_IFREG|0755, st_size=1868984, ...}) = 0
[00007fde8809847a] mmap(NULL, 3971488, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7fde87cb3000
[00007fde88098517] mprotect(0x7fde87e73000, 2097152, PROT_NONE) = 0
[00007fde8809847a] mmap(0x7fde88073000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1c0000) = 0x7fde88073000
[00007fde8809847a] mmap(0x7fde88079000, 14752, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fde88079000
[00007fde88098427] close(3)              = 0
[00007fde8809847a] mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fde88290000
[00007fde8809847a] mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fde8828f000
[00007fde8809847a] mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fde8828e000
[00007fde8807dbd5] arch_prctl(ARCH_SET_FS, 0x7fde8828f700) = 0
[00007fde88098517] mprotect(0x7fde88073000, 16384, PROT_READ) = 0
[00007fde88098517] mprotect(0x600000, 4096, PROT_READ) = 0
[00007fde88098517] mprotect(0x7fde882a2000, 4096, PROT_READ) = 0
[00007fde880984f7] munmap(0x7fde88291000, 58070) = 0
```

기본적인 바이너리 시작 로직은 2가지로 나뉘어 진다.

1. Statically Linked Binary

ex) gcc -o test test.c -static

```
root@9eb26f7bbdb1:/shared/cpp# ldd a.out
    not a dynamic executable

root@9eb26f7bbdb1:/shared/cpp# file a.out
```

```
a.out: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked,
for GNU/Linux 2.6.32, BuildID[sha1]=9f5f0834dea3e988587dd3c553b5aa85b800eb91, not
stripped
```

statically linked binary의 경우에는 shell에서 해당 바이너리를 실행시켰을 때, 크게는 아래의 루틴을 따르게 된다.
./a.out -> fork() -> execve("./a.out", *argv[], *envp[]) 여기까지가 유저레벨에서의 로직
sys_execve() -> do_execve() -> do_execveat_common() -> search_binary_handler() -> load_elf_binary() 여기가 커널내부 로직 (execve를 타고 내부적으로 수행되는 루틴)
_start() -> main() 여기가 execve() 루틴이 실행되고나서 실제 유저레벨에서 실행되는 바이너리 실행부.

```
root@9eb26f7bbdb1:/shared/cpp# readelf -h a.out
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 03 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - GNU
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400890
  Start of program headers:          64 (bytes into file)
  Start of section headers:          910584 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         6
  Size of section headers:           64 (bytes)
  Number of section headers:         33
  Section header string table index: 30
```

Entry Point는 유저레벨에서 가장 먼저 실행하게 되는 바이너리 진입점이다.

```
gdb-peda$ pd _start
Dump of assembler code for function _start:
   0x0000000000400890 <+0>:     xor    ebp,ebp
   0x0000000000400892 <+2>:     mov    r9,rdx
   0x0000000000400895 <+5>:     pop    rsi
   0x0000000000400896 <+6>:     mov    rdx,rsp
   0x0000000000400899 <+9>:     and    rsp,0xfffffffffffffff0
   0x000000000040089d <+13>:    push   rax
   0x000000000040089e <+14>:    push   rsp
   0x000000000040089f <+15>:    mov    r8,0x4015e0
   0x00000000004008a6 <+22>:    mov    rcx,0x401550
   0x00000000004008ad <+29>:    mov    rdi,0x4009ae
   0x00000000004008b4 <+36>:    call   0x400d00 <__libc_start_main>
   0x00000000004008b9 <+41>:    hlt
End of assembler dump.
```

Disassemble 결과를 보면, _start()로직의 주소가 0x400890이므로 Entry Point 지점이랑 일치한다.
일반적으로 많은 사람들이 바이너리가 main부터 실행되는 줄 알지만, 사실은 _start()로직에서 필요한 스택환경이나 환경변수를 세팅하고 __libc_start_main에서 실제 바이너리 내부의 main()을 호출하게 된다.

1. Dynamically Linked Binary

ex) gcc -o test test.c

```
root@9eb26f7bbdb1:/shared/cpp# ldd a.out
    linux-vdso.so.1 =>  (0x00007fff455c8000)
    libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f598de87000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f598dabd000)
    libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f598d7b4000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f598e209000)
    libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f598d59e000)

root@9eb26f7bbdb1:/shared/cpp# file a.out
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=585c3ce8777346adbca0792488cd58d40aa76b3f, not stripped
```

dynamically linked binary의 경우에도 statically linked binary와 대체로 유사한 양상을 띈다.

```
gdb-peda$ pd _start
Dump of assembler code for function _start:
   0x00000000004004c0 <+0>:     xor    ebp,ebp
   0x00000000004004c2 <+2>:     mov    r9,rdx
   0x00000000004004c5 <+5>:     pop    rsi
   0x00000000004004c6 <+6>:     mov    rdx,rsp
   0x00000000004004c9 <+9>:     and    rsp,0xfffffffffffffff0
   0x00000000004004cd <+13>:    push   rax
   0x00000000004004ce <+14>:    push   rsp
   0x00000000004004cf <+15>:    mov    r8,0x400670
   0x00000000004004d6 <+22>:    mov    rcx,0x400600
   0x00000000004004dd <+29>:    mov    rdi,0x4005b6
   0x00000000004004e4 <+36>:    call   0x400480 <__libc_start_main@plt>
   0x00000000004004e9 <+41>:    hlt
End of assembler dump.
```

statically linked냐 dynamically linked냐의 차이는 결국 ld.so의 역할이 있냐 없냐의 차이로 구분지어지게 되고,
바이너리 내부에서는 실제로 .plt, .got section의 역할에 좌우되게 된다.

실제로 실행되는 로직은 아래와 같다.
./a.out -> fork() -> execve("./a.out", *argv[], *envp[]) 여기까지가 유저레벨에서의 로직

sys_execve() -> do_execve() -> do_execveat_common() -> search_binary_handler() -> load_elf_binary() 여기가 커널내부 로직 (execve를 타고 내부적으로 수행되는 루틴)
ld.so -> _start() -> __libc_start_main() -> __init() -> main() 여기가 execve() 루틴이 실행되고나서 실제 유저레벨에서 실행되는 바이너리 실행부.

제일 아래의 부분에서 최종적으로 statically linked와 dynamically linked의 차이가 발생하게 된다.

실제 처음으로 트레이싱되는 execve의 실제 내부 로직은 아래와 같다.

```
gdb-peda$ pd execve
Dump of assembler code for function execve:
   0x000000000043e830 <+0>:    mov    eax,0x3b
   0x000000000043e835 <+5>:    syscall
   0x000000000043e837 <+7>:    cmp    rax,0xfffffffffffff001
   0x000000000043e83d <+13>:    jae    0x444130 <__syscall_error>
   0x000000000043e843 <+19>:    ret
End of assembler dump.
```

execve("/bin/sh", 0, 0);이라는 것을 만약에 실행했다고 가정을 하면, 32bit냐 64bit냐에 따라 인자를 구성해주는 calling convention의 차이는 있겠지만,
해당 실행환경은 64bit였고, 현재 대부분의 리눅스 머신들이 64bit환경에서 돌아가므로 (IoT arm 아키텍쳐가 아니라면,,,) syscall table의 0x3b index에 위치하는
execve()에 대한 시스템 콜을 호출하게 된다.

핵심적으로 분석해야할 것은, kernel 내부의 execve() 로직과 _start함수가 불리기 직전까지의 로직이다.
위에서 나온 fork()와 execve()로 이어지는 첫 로직은 쉘(여기서는 bash shell)에서 실행시켜주는 것이다.

syscall table에 실제로 execve()에 대한 콜을 아래와 같이 wrapping되어 있다.

```
885 asmlinkage long sys_execve(const char __user *filename,
886         const char __user *const __user *argv,
887         const char __user *const __user *envp);
```

asmlinkage라는 prefix를 통해 해당 함수가 어셈블리어와 링크될 수 있음을 나타내고,
실제 구현은 #define asmlinkage CPP_ASMLINKAGE __attribute__((regparm(0)))
이런식으로 되어 있다.
regparm은 몇개의 인자를 레지스터에 담아서 함수를 호출할 지를 내부적으로 알려주는 역할을 하게 된다.

0x3b에 대한 syscall을 요청하게되면, kernel내부 소스의 linux-4.14.11/fs/exec.c 안의 아래의 루틴이 실행된다.

```
1919 SYSCALL_DEFINE3(execve,
```

```
1920         const char __user *, filename,
1921         const char __user *const __user *, argv,
1922         const char __user *const __user *, envp)
1923 {
1924     return do_execve(getname(filename), argv, envp);
1925 }
```

여기서 __user라는 것은 해당하는 변수가 kernel space에 있는 변수가 아니라, user space에 있는 변수임을 의미한다.
getname에서는 결국 filename인 argv[0]이 들어가게 되는데, 이 경로는 ./a.out같이 상대경로가 될 수도 있고, /shared/cpp/a.out처럼 절대 경로가 될 수 있는데
이를 solve해주는 역할을 하게 된다.

이 이후에는 do_execve루틴을 실행하게되는데, 로직은 아래와 같다.

```
1837 int do_execve(struct filename *filename,
1838     const char __user *const __user *__argv,
1839     const char __user *const __user *__envp)
1840 {
1841     struct user_arg_ptr argv = { .ptr.native = __argv };
1842     struct user_arg_ptr envp = { .ptr.native = __envp };
1843     return do_execveat_common(AT_FDCWD, filename, argv, envp, 0);
1844 }
```

linux-4.14.11/fs/exec.c에 위치한 소스이며, do_execveat_common()이라는 함수를 호출한다.

```
1690 static int do_execveat_common(int fd, struct filename *filename,
1691                   struct user_arg_ptr argv,
1692                   struct user_arg_ptr envp,
1693                   int flags)
1694 {
```

linux-4.14.11/fs/exec.c에 동일하게 구현되어 있는 로직이며, 내부적으로 많은 커널루틴을 실행하게 되는데,
무게를 둬서 분석을 해야할 것은 search_binary_handler()와 load_elf_binary() 그리고, ld.so의 역할이다.

do_execveat_common()은 해당 process의 security issue를 체크하게되고, 메모리 관리 구조체를 초기화하는 로직을 타게 된다.
그리고 나서야 search_binary_handler()를 호출하고, 적절한 binary interpreter를 찾기 시작한다.
search_binary_handler()는 동일 소스상의 조금 아래의 위치에 있는 exec_binprm()에서 호출하게 된다.

```
1798     retval = exec_binprm(bprm);
1799     if (retval < 0)
1800         goto out;
```

bprm은 struct linux_binprm 구조체로, 실제 구현부는 linux-4.14.11/include/linux/binfmts.h 아래에 존재한다.

```
17 struct linux_binprm {
18     char buf[BINPRM_BUF_SIZE];
19 #ifdef CONFIG_MMU
20     struct vm_area_struct *vma;
21     unsigned long vma_pages;
22 #else
23 # define MAX_ARG_PAGES  32
24     struct page *page[MAX_ARG_PAGES];
25 #endif
26     struct mm_struct *mm;

...
```

실제 binary를 loading할 때, 사용되는 인자들을 가지고 있으며, virtual memory mapping에 대한 정보도 가지게 된다.
해당 구조체 정보를 가지고 exec_binprm(bprm);을 호출하는데 내부 구현은 아래와 같이 되어 있다.

```
1665 static int exec_binprm(struct linux_binprm *bprm)
1666 {
1667     pid_t old_pid, old_vpid;
1668     int ret;
1669
1670     /* Need to fetch pid before load_binary changes it */
1671     old_pid = current->pid;
1672     rcu_read_lock();
1673     old_vpid = task_pid_nr_ns(current, task_active_pid_ns(current->parent));
1674     rcu_read_unlock();
1675
1676     ret = search_binary_handler(bprm);
1677     if (ret >= 0) {
1678         audit_bprm(bprm);
1679         trace_sched_process_exec(current, old_pid, bprm);
1680         ptrace_event(PTRACE_EVENT_EXEC, old_vpid);
1681         proc_exec_connector(current);
1682     }
1683
1684     return ret;
1685 }
```

binary를 실행하고 라이브러리를 로딩할 수 있는 핸들러는 1676 line의 search_binary_handler에서 진행하게 된다.

```
1612 int search_binary_handler(struct linux_binprm *bprm)
1613 {
1614     bool need_retry = IS_ENABLED(CONFIG_MODULES);
1615     struct linux_binfmt *fmt;
1616     int retval;
1617
```

```
1618     /* This allows 4 levels of binfmt rewrites before failing hard. */
1619     if (bprm->recursion_depth > 5)
1620         return -ELOOP;
```

동일하게 linux-4.14.11/fs/exec.c에 위치한 소스이다.
해당하는 binary를 실행할 수 있는 handler를 list_for_each 커널내부 매크로를 통해 찾게된다.
Handler는 해당 바이너리의 처음에 위치한 magic byte를 바탕으로 결정되게 된다. (일반적인 Linux의 바이너리의 경우 : \x7fELF)
만약 호출할 수 있는 handler를 찾지 못한다면, kerneld를 통해 새로운 handler를 load할려고 하며, 여기서도 로드가 불가능하게되면
ENOEXEC라는 "Exec format error"를 보내게 된다.

```
1622     retval = security_bprm_check(bprm);
```

1622라인의 sanity check 루틴을 거친 이후에 list_for_each_entry()매크로를 진행하게 된다.
커널 분석을 함에 있어서, 내부적으로 많은 double-linked-list 자료구조를 사용하게 되는데,
이를 효율적으로 탐색하기 위한 매크로가 존재하는데
그것이 list_for_each_entry이다.

```
457 /**
458  * list_for_each_entry  -   iterate over list of given type
459  * @pos:    the type * to use as a loop cursor.
460  * @head:   the head for your list.
461  * @member: the name of the list_head within the struct.
462  */
463 #define list_for_each_entry(pos, head, member)              \
464     for (pos = list_first_entry(head, typeof(*pos), member);    \
465          &pos->member != (head);                    \
466          pos = list_next_entry(pos, member))
```

list_for_each_entry는 linux-4.14.11/include/linux/list.h에 정의되어 있다.

```
1629     list_for_each_entry(fmt, &formats, lh) {
1630         if (!try_module_get(fmt->module))
1631             continue;
1632         read_unlock(&binfmt_lock);
1633         bprm->recursion_depth++;
1634         retval = fmt->load_binary(bprm)
```

fmt는 iterator로 역할을 하며, &formats는 순회하기 원하는 리스트의 포인터, 마지막 lh는 list_head를 의미한다.
formats 배열을 순회하기 시작하며, linux_binfmt 객체인 멤버를 읽을 수 있게 한다. 그리고 load_binary필드를 읽는다.
그리고 해당 필드에 있는 load_binary()를 호출하며, 리턴값을 확인하게 된다.
즉, formats는 일종의 등록된 바이너리 인터프리터에 대한 정보를 가지고있는 테이블이라고 볼 수 있다.
ELF에서 호출가능한 핸들러는 linux-4.14.11/fs/binfmt_elf.c에 정의되어 있다.

```
 88 static struct linux_binfmt elf_format = {
 89     .module     = THIS_MODULE,
 90     .load_binary    = load_elf_binary,
 91     .load_shlib = load_elf_library,
 92     .core_dump  = elf_core_dump,
 93     .min_coredump   = ELF_EXEC_PAGESIZE,
 94 };
```

같은 function의 아랫 부분이다.
흔히들 c는 절차지향, c++은 객체지향이라지만, c에서도 객체를 구현할 수 있으며, 구조체 내부의 함수포인터를 통해 임의의 virtual함수를 구현할 수도 있다.
위의 fmt->load_binary의 경우가 그러하다.
해당하는 loader가 있게되면, 바이너리를 실행할 수 있는 루틴을 호출하게 된다.
fmt라는 것은 struct linux_binfmt 커널내부 구조체로 아래와 같은 형태를 가지게 된다.

```
92 struct linux_binfmt {
93     struct list_head lh;
94     struct module *module;
95     int (*load_binary)(struct linux_binprm *);
96     int (*load_shlib)(struct file *);
97     int (*core_dump)(struct coredump_params *cprm);
98     unsigned long min_coredump; /* minimal dump size */
99 } __randomize_layout;
```

load_binary는 binary를 execute하기 위한 method(function pointer)이고, load_shlib의 경우엔 shared library를 loading하는 method(function), 마지막으로
core_dump는 core파일을 생성하기위한 method(function)이다.

전반부에서 실행하였던 a.out의 경우에는 ELF format이므로 아래의 load_elf_binary핸들러가 호출되게 된다.
load_elf_binary()의 구현부는 아래와 같다.

```
679 static int load_elf_binary(struct linux_binprm *bprm)
680 {
681     struct file *interpreter = NULL; /* to shut gcc up */
682     unsigned long load_addr = 0, load_bias = 0;
683     int load_addr_set = 0;
684     char * elf_interpreter = NULL;
685     unsigned long error;

...

868     setup_new_exec(bprm);
869     install_exec_creds(bprm);

...
```

```
1063     if (elf_interpreter) {
1064         unsigned long interp_map_addr = 0;
1065
1066         elf_entry = load_elf_interp(&loc->interp_elf_ex,
1067                         interpreter,
1068                         &interp_map_addr,
1069                         load_bias, interp_elf_phdata);

...

1112     current->mm->end_code = end_code;
1113     current->mm->start_code = start_code;
1114     current->mm->start_data = start_data;
1115     current->mm->end_data = end_data;
1116     current->mm->start_stack = bprm->p;

...

1149     start_thread(regs, elf_entry, bprm->p);
```

linux-4.14.11/fs/binfmt_elf.c에 위치해있다.
이 소스파일에서는 elf와 관련된 핸들러를 구현하고 있다.
868~869 line에서 새롭게 실행할 프로그램의 커널 내부 상태와, credential을 설정하게 된다.
1063~1069 line은 ELF interpreter를 메모리에 올리게 되는데, 이는 이후 execve()가 끝난 이후에 실행되게 된다.
1112~1116 line을 통해 실제 메모리 매핑이 이루어지게 되고,
load_elf_binary()는 1149 line의 start_thread()를 통해 실제로 해당 파일이 스케줄링되기 시작한다.
뿐만 아니라, library를 호출하는 함수도 동일한 파일내에 구현되어 있다.

```
1172 static int load_elf_library(struct file *file)
1173 {
1174     struct elf_phdr *elf_phdata;
1175     struct elf_phdr *eppnt;
1176     unsigned long elf_bss, bss, len;
1177     int retval, error, i, j;
1178     struct elfhdr elf_ex;
1179     loff_t pos = 0;
1180
1181     error = -ENOEXEC;
1182     retval = kernel_read(file, &elf_ex, sizeof(elf_ex), &pos);
...
```

load_elf_library()는 해당 ELF file의 헤더부분을 읽어서 a.out에 관련된 library를 올리게 된다.
그리고 커널은 binary의 INTERP header를 읽어서 dynamic loader를 호출하게 된다.

```
367 void
368 start_thread(struct pt_regs *regs, unsigned long new_ip, unsigned long new_sp)
369 {
370     start_thread_common(regs, new_ip, new_sp,
371                 __USER_CS, __USER_DS, 0);
372 }
```

64bit 환경에서의 start_thread()는 linux-4.14.11/arch/x86/kernel/process_64.c에 구현되어 있다.
인자는 순서대로 아래와 같다.

- 새로운 task에 대한 레지스터
- 새로운 task에 대한 entry point
- 새로운 task의 stack top

```
341 static void
342 start_thread_common(struct pt_regs *regs, unsigned long new_ip,
343             unsigned long new_sp,
344             unsigned int _cs, unsigned int _ss, unsigned int _ds)
345 {
346     WARN_ON_ONCE(regs != current_pt_regs());
347
348     if (static_cpu_has(X86_BUG_NULL_SEG)) {
349         /* Loading zero below won't clear the base. */
350         loadsegment(fs, __USER_DS);
351         load_gs_index(__USER_DS);
352     }
353
354     loadsegment(fs, 0);
355     loadsegment(es, _ds);
356     loadsegment(ds, _ds);
357     load_gs_index(0);
358
359     regs->ip        = new_ip;
360     regs->sp        = new_sp;
361     regs->cs        = _cs;
362     regs->ss        = _ss;
363     regs->flags     = X86_EFLAGS_IF;
364     force_iret();
365 }
```

같은 파일내에 start_thread_common()도 같이 구현되어 있다.
세그먼트 레지스터나 일반적인 레지스터의 값을 초기화시켜주고, force_iret()매크로를 통해 iret (interrupt return) instruction을 호출한다.
여기까지 진행함으로써, user space에서 해당 바이너리를 실행시킬 모든 준비를 다하였고, execve()가 끝이난다.

아래부터는 유저스페이스에서의 로더쪽에 관한 설명이다.

```
gdb-peda$ vmmap
Start              End                Perm    Name
0x00400000         0x00401000         r-xp    /shared/cpp/a.out
0x00600000         0x00601000         r--p    /shared/cpp/a.out
0x00601000         0x00602000         rw-p    /shared/cpp/a.out
0x00007ffff7a0d000 0x00007ffff7bcd000 r-xp    /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7bcd000 0x00007ffff7dcd000 ---p    /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dcd000 0x00007ffff7dd1000 r--p    /lib/x86_64-linux-gnu/libc-2.23.so
```

```
0x00007ffff7dd1000 0x00007ffff7dd3000 rw-p    /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd3000 0x00007ffff7dd7000 rw-p    mapped
0x00007ffff7dd7000 0x00007ffff7dfd000 r-xp    /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7fe4000 0x00007ffff7fe7000 rw-p    mapped
0x00007ffff7ff6000 0x00007ffff7ff8000 rw-p    mapped
0x00007ffff7ff8000 0x00007ffff7ffa000 r--p    [vvar]
0x00007ffff7ffa000 0x00007ffff7ffc000 r-xp    [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 r--p    /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffd000 0x00007ffff7ffe000 rw-p    /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffe000 0x00007ffff7fff000 rw-p    mapped
0x00007ffffffde000 0x00007ffffffff000 rw-p    [stack]
0xffffffffff600000 0xffffffffff601000 r-xp    [vsyscall]
```

실제로 바이너리를 실행하고 user space에서 virtual memory를 보게되면, ld-2.23.so같은 로더를 메모리에 로드한다.
로더를 커널이 메모리에 올리고나면, ELF파일이 메모리에 실행되게 된다.
로더의 역할이 시작되게되면 여기서부터는 kernel space 로직이 아닌, user space에서의 역할이 시작되는 것이다.

kernel내부 로직에서 메모리 관리 구조체를 초기화하는 부분이 있다고 했었는데, 그것은 아래의 정보들을 이용하여 virtual memory에 mapping하게 된다.
readelf -l 을 통해 매핑되는 영역을 보면 아래와 같다.
Binary자체는 실행하지 않았을 때는, 파일 그 자체이므로, 디스크에 있는 것이지만, 실행하게 되면, memory에 올라가야하므로,
Code, Data, Heap, Stack segement에 각각 매핑을 시켜줘야 한다.

```
root@9eb26f7bbdb1:/shared/cpp# readelf -l a.out

Elf file type is EXEC (Executable file)
Entry point 0x400890
There are 6 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000c90c7 0x00000000000c90c7  R E    200000
  LOAD           0x00000000000c9eb8 0x00000000006c9eb8 0x00000000006c9eb8
                 0x0000000000001c98 0x0000000000003550  RW     200000
  NOTE           0x0000000000000190 0x0000000000400190 0x0000000000400190
                 0x0000000000000044 0x0000000000000044  R      4
  TLS            0x00000000000c9eb8 0x00000000006c9eb8 0x00000000006c9eb8
                 0x0000000000000020 0x0000000000000050  R      8
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x00000000000c9eb8 0x00000000006c9eb8 0x00000000006c9eb8
                 0x0000000000000148 0x0000000000000148  R      1

Section to Segment mapping:
  Segment Sections...
```

```
   00     .note.ABI-tag .note.gnu.build-id .rela.plt .init .plt .text
__libc_freeres_fn __libc_thread_freeres_fn .fini .rodata __libc_subfreeres
__libc_atexit .stapsdt.base __libc_thread_subfreeres .eh_frame .gcc_except_table
   01     .tdata .init_array .fini_array .jcr .data.rel.ro .got .got.plt .data .bss
__libc_freeres_ptrs
   02     .note.ABI-tag .note.gnu.build-id
   03     .tdata .tbss
   04
   05     .tdata .init_array .fini_array .jcr .data.rel.ro .got
```

로더는 ELF header를 parsing하고 dlopen()을 호출한다. (library function)
dlopen을 통해 dynamically linked binary에서 필요한 shared library들을 memory에 올리게 된다.
그리고 _start()함수를 시작하면서 본격적으로 해당 바이너리가 실행되게 된다.

여기서부터는 라이브러리(glibc)의 내부 로더의 구현을 따르게 된다.

```
root@9eb26f7bbdb1:/shared/cpp# strace -fFi ./a.out
[00007f58b41b8777] execve("./a.out", ["./a.out"], [/* 17 vars */]) = 0
[00007f498cf494b9] brk(NULL)             = 0x2182000
[00007f498cf4a387] access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or
directory)
[00007f498cf4a47a] mmap(NULL, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f498d152000
[00007f498cf4a387] access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or
directory)
[00007f498cf4a327] open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
[00007f498cf4a2b4] fstat(3, {st_mode=S_IFREG|0644, st_size=58070, ...}) = 0
[00007f498cf4a47a] mmap(NULL, 58070, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f498d143000
[00007f498cf4a427] close(3)              = 0
[00007f498cf4a387] access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or
directory)
[00007f498cf4a327] open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
[00007f498cf4a347] read(3,
"\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"..., 832) = 832
[00007f498cf4a2b4] fstat(3, {st_mode=S_IFREG|0755, st_size=1868984, ...}) = 0
[00007f498cf4a47a] mmap(NULL, 3971488, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f498cb65000
[00007f498cf4a517] mprotect(0x7f498cd25000, 2097152, PROT_NONE) = 0
[00007f498cf4a47a] mmap(0x7f498cf25000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1c0000) = 0x7f498cf25000
[00007f498cf4a47a] mmap(0x7f498cf2b000, 14752, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f498cf2b000
[00007f498cf4a427] close(3)              = 0
[00007f498cf4a47a] mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f498d142000
[00007f498cf4a47a] mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f498d141000
[00007f498cf4a47a] mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f498d140000
[00007f498cf2fbd5] arch_prctl(ARCH_SET_FS, 0x7f498d141700) = 0
[00007f498cf4a517] mprotect(0x7f498cf25000, 16384, PROT_READ) = 0
```

```
[00007f498cf4a517] mprotect(0x600000, 4096, PROT_READ) = 0
[00007f498cf4a517] mprotect(0x7f498d154000, 4096, PROT_READ) = 0
[00007f498cf4a4f7] munmap(0x7f498d143000, 58070) = 0


root@9eb26f7bbdb1:/# cat /proc/10833/maps
00400000-00401000 r-xp 00000000 00:46 5781750
/shared/cpp/a.out
00600000-00601000 r--p 00000000 00:46 5781750
/shared/cpp/a.out
00601000-00602000 rw-p 00001000 00:46 5781750
/shared/cpp/a.out
02182000-021a3000 rw-p 00000000 00:00 0                                  [heap]
7f498cb65000-7f498cd25000 r-xp 00000000 08:01 2229385
/lib/x86_64-linux-gnu/libc-2.23.so
7f498cd25000-7f498cf25000 ---p 001c0000 08:01 2229385
/lib/x86_64-linux-gnu/libc-2.23.so
7f498cf25000-7f498cf29000 r--p 001c0000 08:01 2229385
/lib/x86_64-linux-gnu/libc-2.23.so
7f498cf29000-7f498cf2b000 rw-p 001c4000 08:01 2229385
/lib/x86_64-linux-gnu/libc-2.23.so
7f498cf2b000-7f498cf2f000 rw-p 00000000 00:00 0
7f498cf2f000-7f498cf55000 r-xp 00000000 08:01 2229372
/lib/x86_64-linux-gnu/ld-2.23.so
7f498d140000-7f498d143000 rw-p 00000000 00:00 0
7f498d152000-7f498d154000 rw-p 00000000 00:00 0
7f498d154000-7f498d155000 r--p 00025000 08:01 2229372
/lib/x86_64-linux-gnu/ld-2.23.so
7f498d155000-7f498d156000 rw-p 00026000 08:01 2229372
/lib/x86_64-linux-gnu/ld-2.23.so
7f498d156000-7f498d157000 rw-p 00000000 00:00 0
7ffe78a30000-7ffe78a51000 rw-p 00000000 00:00 0                          [stack]
7ffe78b2e000-7ffe78b30000 r--p 00000000 00:00 0                          [vvar]
7ffe78b30000-7ffe78b32000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

다시 한번 strace를 통해 mmap된 리턴 값들과, 해당 프로세스가 실제로 사용중인 virtual memory maps를 보게되면, 라이브러리 path를 찾아서
open하고, open된 파일디스크립터를 바탕으로 mmap함수를 통해 메모리에 매핑하게 된다.

strace는 user -> kernel로의 system call / signal을 trace하는 도구이므로, 커널내부 로직이 아닌 유저랜드에서의 호출이 찍힌 것이다.
execve가 돌아오고 나서부터는 _start()함수가 시작되므로, 이후 strace가 남긴 로그는 _start()이후인 __libc_start_main()에서부터 찍힌 것이라고
볼 수 있다.

dynamically linked된 binary의 경우에는 .plt와 .got와 같은 섹션을 사용하며, lazy binding이라는 기법을 이용하기 때문에,주로 runtime에 해당하는 함수의 매핑된 라이브러리 상의 주소를 구해온다.

```
44 int
45 __libc_start_main (int argc, char **argv,
```

```
46            char **ev,
47            ElfW (auxv_t) * auxvec,
48            void (*rtld_fini) (void),
49            struct startup_info *stinfo,
50            char **stack_on_entry)
51 {
```

실제 __libc_start_main()은 glibc-2.26/sysdeps/unix/sysv/linux/powerpc/libc-start.c에서도 찾아볼 수 있지만, 최신 라이브러리에서는 __libc_main_start호출 로직같은 것은 start.S라는 어셈블리 파일에서 다루고 있다.

```
105 #else
106     /* Pass address of our own entry points to .fini and .init.  */
107     mov $__libc_csu_fini, %R8_LP
108     mov $__libc_csu_init, %RCX_LP
109
110     mov $main, %RDI_LP
111 #endif
112
113     /* Call the user's main function, and exit with its value.
114        But let the libc call main.  Since __libc_start_main in
115        libc.so is called very early, lazy binding isn't relevant
116        here.  Use indirect branch via GOT to avoid extra branch
117        to PLT slot.  In case of static executable, ld in binutils
118        2.26 or above can convert indirect branch into direct
119        branch.  */
120     call *__libc_start_main@GOTPCREL(%rip)
121
122     hlt         /* Crash if somehow `exit' does return.  */
123 END (_start)
```

그리고 strace의 이후 찍히게 되는 로그는 __libc_start_main이 아닌, rtld쪽이므로, 디버깅을 통해 해당 호출위치를 알 수 있다.

```
#0  0x00007ffff7df247a in __mmap (addr=addr@entry=0x0, len=0xe2d6,
prot=prot@entry=0x1, flags=flags@entry=0x2, fd=fd@entry=0x3,
offset=offset@entry=0x0)
    at ../sysdeps/unix/sysv/linux/wordsize-64/mmap.c:34
#1  0x00007ffff7de8775 in _dl_sysdep_read_whole_file
(file=file@entry=0x7ffff7df7232 "/etc/ld.so.cache",
sizep=sizep@entry=0x7ffff7ffe0a0 <cachesize>, prot=prot@entry=0x1) at dl-misc.c:62
#2  0x00007ffff7def508 in _dl_load_cache_lookup (name=name@entry=0x400391
"libc.so.6") at dl-cache.c:199
#3  0x00007ffff7de0169 in _dl_map_object (loader=0x7ffff7ffe168, name=0x400391
"libc.so.6", type=0x1, trace_mode=0x0, mode=<optimized out>, nsid=<optimized out>)
at dl-load.c:2342
#4  0x00007ffff7de4ba2 in openaux (a=a@entry=0x7fffffffe140) at dl-deps.c:63
#5  0x00007ffff7de7564 in _dl_catch_error (objname=objname@entry=0x7fffffffe138,
errstring=errstring@entry=0x7fffffffe130, mallocedp=mallocedp@entry=0x7fffffffe12f,
    operate=operate@entry=0x7ffff7de4b70 <openaux>, args=args@entry=0x7fffffffe140)
at dl-error.c:187
```

```
#6  0x00007ffff7de51e2 in _dl_map_object_deps (map=map@entry=0x7ffff7ffe168,
preloads=<optimized out>, npreloads=npreloads@entry=0x0,
trace_mode=trace_mode@entry=0x0, open_mode=open_mode@entry=0x0)
    at dl-deps.c:254
#7  0x00007ffff7ddaa29 in dl_main (phdr=<optimized out>, phnum=<optimized out>,
user_entry=<optimized out>, auxv=<optimized out>) at rtld.c:1647
#8  0x00007ffff7df0632 in _dl_sysdep_start
(start_argptr=start_argptr@entry=0x7fffffffe390,
dl_main=dl_main@entry=0x7ffff7dd91e0 <dl_main>) at ../elf/dl-sysdep.c:249
#9  0x00007ffff7dd8c2a in _dl_start_final (arg=0x7fffffffe390) at rtld.c:323
#10 _dl_start (arg=0x7fffffffe390) at rtld.c:429
#11 0x00007ffff7dd7c38 in _start () from /lib64/ld-linux-x86-64.so.2
#12 0x0000000000000001 in ?? ()
#13 0x00007fffffffe592 in ?? ()
#14 0x0000000000000000 in ?? ()
```

gdb를 통해 syscall이 호출될 때, 자동으로 브레이크를 잡아서 볼 수가 있는데, mmap이 처음불리는 시점을 잡아서 콜스택을 분석해보면 위와 같다.
_start()호출 이후에, dynamic loader내부에서 특정 라이브러리나 특정 파일들에 대하여 mmap을 하는 것을 알 수 있다.

open의 경우도 아래와 같이 호출을 하게 된다.

```
Starting program: /shared/cpp/a.out
[----------------------------------registers-----------------------------------]
RAX: 0xffffffffffffffda
RBX: 0x400391 ("libc.so.6")
RCX: 0x7ffff7df2327 (<open64+7>:    cmp    rax,0xfffffffffffff001)
RDX: 0x1
RSI: 0x80000
RDI: 0x7ffff7df7232 ("/etc/ld.so.cache")
RBP: 0xffffffffffffffff
RSP: 0x7fffffffd9a8 --> 0x7ffff7de8716 (<_dl_sysdep_read_whole_file+38>:    test
eax,eax)
RIP: 0x7ffff7df2327 (<open64+7>:    cmp    rax,0xfffffffffffff001)
R8 : 0x0
R9 : 0x0
R10: 0x7ffff7ffe480 --> 0xffffffffffffffff
R11: 0x202
R12: 0x7ffff7ffe0a0 --> 0x0
R13: 0x1
R14: 0x7ffff7ffd040 --> 0x7ffff7ffe168 --> 0x0
R15: 0x7ffff7ffe510 --> 0xffffffffffffffff
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x7ffff7df231f <__GI___lxstat+63>:    ret
   0x7ffff7df2320 <open64>:    mov    eax,0x2
   0x7ffff7df2325 <open64+5>:    syscall
=> 0x7ffff7df2327 <open64+7>:    cmp    rax,0xfffffffffffff001
   0x7ffff7df232d <open64+13>:    jae    0x7ffff7df2330 <open64+16>
   0x7ffff7df232f <open64+15>:    ret
   0x7ffff7df2330 <open64+16>:    lea    rcx,[rip+0x20bde9]        # 0x7ffff7ffe120
<rtld_errno>
```

```
   0x7ffff7df2337 <open64+23>:    neg    eax
[-------------------------------------stack--------------------------------------]
0000| 0x7fffffffd9a8 --> 0x7ffff7de8716 (<_dl_sysdep_read_whole_file+38>:    test
eax,eax)
0008| 0x7fffffffd9b0 --> 0x0
0016| 0x7fffffffd9b8 --> 0x0
0024| 0x7fffffffd9c0 --> 0x0
0032| 0x7fffffffd9c8 --> 0x0
0040| 0x7fffffffd9d0 --> 0x0
0048| 0x7fffffffd9d8 --> 0x0
0056| 0x7fffffffd9e0 --> 0x0
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value

Catchpoint 1 (call to syscall open), 0x00007ffff7df2327 in open64 () at
../sysdeps/unix/syscall-template.S:84
84    ../sysdeps/unix/syscall-template.S: No such file or directory.
gdb-peda$ bt
#0  0x00007ffff7df2327 in open64 () at ../sysdeps/unix/syscall-template.S:84
#1  0x00007ffff7de8716 in _dl_sysdep_read_whole_file
(file=file@entry=0x7ffff7df7232 "/etc/ld.so.cache",
sizep=sizep@entry=0x7ffff7ffe0a0 <cachesize>, prot=prot@entry=0x1) at dl-misc.c:52
#2  0x00007ffff7def508 in _dl_load_cache_lookup (name=name@entry=0x400391
"libc.so.6") at dl-cache.c:199
#3  0x00007ffff7de0169 in _dl_map_object (loader=0x7ffff7ffe168, name=0x400391
"libc.so.6", type=0x1, trace_mode=0x0, mode=<optimized out>, nsid=<optimized out>)
at dl-load.c:2342
#4  0x00007ffff7de4ba2 in openaux (a=a@entry=0x7fffffffe140) at dl-deps.c:63
#5  0x00007ffff7de7564 in _dl_catch_error (objname=objname@entry=0x7fffffffe138,
errstring=errstring@entry=0x7fffffffe130, mallocedp=mallocedp@entry=0x7fffffffe12f,
    operate=operate@entry=0x7ffff7de4b70 <openaux>, args=args@entry=0x7fffffffe140)
at dl-error.c:187
#6  0x00007ffff7de51e2 in _dl_map_object_deps (map=map@entry=0x7ffff7ffe168,
preloads=<optimized out>, npreloads=npreloads@entry=0x0,
trace_mode=trace_mode@entry=0x0, open_mode=open_mode@entry=0x0)
    at dl-deps.c:254
#7  0x00007ffff7ddaa29 in dl_main (phdr=<optimized out>, phnum=<optimized out>,
user_entry=<optimized out>, auxv=<optimized out>) at rtld.c:1647
#8  0x00007ffff7df0632 in _dl_sysdep_start
(start_argptr=start_argptr@entry=0x7fffffffe390,
dl_main=dl_main@entry=0x7ffff7dd91e0 <dl_main>) at ../elf/dl-sysdep.c:249
#9  0x00007ffff7dd8c2a in _dl_start_final (arg=0x7fffffffe390) at rtld.c:323
#10 _dl_start (arg=0x7fffffffe390) at rtld.c:429
#11 0x00007ffff7dd7c38 in _start () from /lib64/ld-linux-x86-64.so.2
#12 0x0000000000000001 in ?? ()
#13 0x00007fffffffe592 in ?? ()
#14 0x0000000000000000 in ?? ()
```

위에서 strace에서 찍은대로 /etc/ld.so.cache를 open한다.

```
42 void *
43 internal_function
44 _dl_sysdep_read_whole_file (const char *file, size_t *sizep, int prot)
45 {
```

```
46   void *result = MAP_FAILED;
47   struct stat64 st;
48   int fd = __open (file, O_RDONLY | O_CLOEXEC);
49   if (fd >= 0)
50     {
51       if (__fxstat64 (_STAT_VER, fd, &st) >= 0)
52     {
53       *sizep = st.st_size;
54
55       /* No need to map the file if it is empty.  */
56       if (*sizep != 0)
57         /* Map a copy of the file contents.  */
58         result = __mmap (NULL, *sizep, prot,
59 #ifdef MAP_COPY
60                  MAP_COPY
61 #else
62                  MAP_PRIVATE
63 #endif
64 #ifdef MAP_FILE
65                  | MAP_FILE
66 #endif
67                  , fd, 0);
68     }
69       __close (fd);
70     }
71   return result;
72 }
```

open을 부르는 함수인 _dl_sysdep_read_whole_file이다.
이 함수는 glibc-2.26/elf/dl-misc.c에 위치하고 있으며, 이런식으로 파일을 오픈하고, __mmap을 통해 해당 파일을 매핑하게 된다.

open -> read -> fstat -> mmap -> mprotect로 이어지는 부분은 아래의 callstack분석으로 찾아낼 수 있다.

```
gdb-peda$ bt
#0  0x00007ffff7df2347 in read () at ../sysdeps/unix/syscall-template.S:84
#1  0x00007ffff7ddc7ab in open_verify (name=name@entry=0x7ffff7ff74a0
"/lib/x86_64-linux-gnu/libc.so.6", fbp=fbp@entry=0x7fffffffddd0, loader=<optimized
out>, whatcode=whatcode@entry=0x8,
    mode=mode@entry=0x0, found_other_class=found_other_class@entry=0x7fffffffddbf,
free_name=0x0, fd=0x3) at dl-load.c:1783
#2  0x00007ffff7de01b6 in _dl_map_object (loader=0x7ffff7ffe168, name=0x400391
"libc.so.6", type=0x1, trace_mode=0x0, mode=<optimized out>, nsid=<optimized out>)
at dl-load.c:2379
#3  0x00007ffff7de4ba2 in openaux (a=a@entry=0x7fffffffe3b0) at dl-deps.c:63
#4  0x00007ffff7de7564 in _dl_catch_error (objname=objname@entry=0x7fffffffe3a8,
errstring=errstring@entry=0x7fffffffe3a0, mallocedp=mallocedp@entry=0x7fffffffe39f,
    operate=operate@entry=0x7ffff7de4b70 <openaux>, args=args@entry=0x7fffffffe3b0)
at dl-error.c:187
#5  0x00007ffff7de51e2 in _dl_map_object_deps (map=map@entry=0x7ffff7ffe168,
preloads=<optimized out>, npreloads=npreloads@entry=0x0,
trace_mode=trace_mode@entry=0x0, open_mode=open_mode@entry=0x0)
```

```
    at dl-deps.c:254
#6  0x00007ffff7ddaa29 in dl_main (phdr=<optimized out>, phnum=<optimized out>,
user_entry=<optimized out>, auxv=<optimized out>) at rtld.c:1647
#7  0x00007ffff7df0632 in _dl_sysdep_start
(start_argptr=start_argptr@entry=0x7fffffffe600,
dl_main=dl_main@entry=0x7ffff7dd91e0 <dl_main>) at ../elf/dl-sysdep.c:249
#8  0x00007ffff7dd8c2a in _dl_start_final (arg=0x7fffffffe600) at rtld.c:323
#9  _dl_start (arg=0x7fffffffe600) at rtld.c:429
#10 0x00007ffff7dd7c38 in _start () from /lib64/ld-linux-x86-64.so.2
#11 0x0000000000000001 in ?? ()
#12 0x00007fffffffe80e in ?? ()
#13 0x0000000000000000 in ?? ()
```

_start() -> _dl_start() -> _dl_start_final() -> _dl_sysdep_start() -> dl_main() -> _dl_map_object_deps() -> _dl_catch_error() -> _dl_map_object() -> open_verify()의 함수호출 로직을 타게된다.

```
2048              fd = open_verify (realname, fd,
2049                                &fb, loader ?: GL(dl_ns)[nsid]._ns_loaded,
2050                                LA_SER_CONFIG, mode, &found_other_class,
2051                                false);
```

open_verify()함수의 경우에는 glibc-2.26/elf/dl-load.c의 _dl_map_object()내부에서 호출된다.
위의 코드는 _dl_map_object()내부에서 open_verify()를 호출하는 코드이다.

```
1522     fd = __open (name, O_RDONLY | O_CLOEXEC);

...

1538       do
1539     {
1540       ssize_t retlen = __libc_read (fd, fbp->buf + fbp->len,
1541                     sizeof (fbp->buf) - fbp->len);
1542       if (retlen <= 0)
1543         break;
1544       fbp->len += retlen;
1545     }
1546       while (__glibc_unlikely (fbp->len < sizeof (ElfW(Ehdr))));
```

동일하게 open_verify()는 glibc-2.26/elf/dl-load.c에 구현되어 있으며, open을 하게되고, read를 통해 ELF header를 읽어오게 된다.
읽어온 ELF header를 통해 valid한 header인지에 대한 검증이 이루어지고되고 다시 _dl_map_object()로 리턴하게 된다.

```
2199   return _dl_map_object_from_fd (name, origname, fd, &fb, realname, loader,
2200                   type, mode, &stack_end, nsid);
```

_dl_map_object()에서 마지막 부분에서 _dl_map_object_from_fd()를 호출하게 되는데, 여기서 _dl_get_file_id()를 호출하게 된다.

```
886   if (__glibc_unlikely (!_dl_get_file_id (fd, &id)))
887     {
```

동일한 dl-load.c에 위치해있으며, 해당 함수 내부에서 fstat()를 호출한다.

```
32 static inline bool
33 _dl_get_file_id (int fd, struct r_file_id *id)
34 {
35   struct stat64 st;
36
37   if (__glibc_unlikely (__fxstat64 (_STAT_VER, fd, &st) < 0))
38     return false;
39
40   id->dev = st.st_dev;
41   id->ino = st.st_ino;
42   return true;
43 }
```

_dl_get_file_id()는 glibc-2.26/sysdeps/posix/dl-fileid.h에 위치해있다.

```
1183     errstring = _dl_map_segments (l, fd, header, type, loadcmds, nloadcmds,
1184                   maplength, has_holes, loader);
```

이후, _dl_map_object_from_fd()로 다시 돌아와서, 해당 함수의 아래쪽 부분의 _dl_map_segments()을 호출한다.

```
92       if (c->mapend > c->mapstart
93           /* Map the segment contents from the file.  */
94           && (__mmap ((void *) (l->l_addr + c->mapstart),
95                       c->mapend - c->mapstart, c->prot,
96                       MAP_FIXED|MAP_COPY|MAP_FILE,
97                       fd, c->mapoff)
98               == MAP_FAILED))
99         return DL_MAP_SEGMENTS_ERROR_MAP_SEGMENT;

...


126                   if (__mprotect ((caddr_t) (zero
127                                              & ~(GLRO(dl_pagesize) - 1)),
128                                   GLRO(dl_pagesize), c->prot|PROT_WRITE) < 0)
129                     return DL_MAP_SEGMENTS_ERROR_MPROTECT;
```

해당 부분에서 실제로 공유 라이브러리에 대한 mapping이 이루어지게되며, strace에 찍혔던 mmap -> mprotect가 여기서 실행되게 된다.
이후의 mmap의 일부는 동일하게 _dl_map_segements()에서 호출되어지며,

```
[00007f498cf4a47a] mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f498d142000
[00007f498cf4a47a] mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f498d141000
[00007f498cf4a47a] mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f498d140000
[00007f498cf2fbd5] arch_prctl(ARCH_SET_FS, 0x7f498d141700) = 0
```

이 부분의 mmap은 dl_main() -> init_tls()에서 호출되게 된다.
init_tls()에서 calloc()을 호출하게되고, calloc은 내부적으로 malloc()을 사용중이며,
malloc()에서 특정 조건을 만족하면 brk syscall이 아닌 mmap syscall을 사용하게 되는데,
여기서 mmap이 사용되는 것이다.

```
695   GL(dl_tls_dtv_slotinfo_list) = (struct dtv_slotinfo_list *)
696     calloc (sizeof (struct dtv_slotinfo_list)
697         + nelem * sizeof (struct dtv_slotinfo), 1);
```

해당 루틴은 glibc-2.26/elf/rtld.c에 위치하고 있다.

이후 수행되는 mprotect는 권한 수정을 위해 dl_main()의 _dl_protect_relro()에서 사용되게 된다.

```
321   if (start != end
322       && __mprotect ((void *) start, end - start, PROT_READ) < 0)
323     {
```

소스는 glibc-2.26/elf/dl-reloc.c에 위치하고 있다.
그리고 마지막으로 불리는 munmap()은 dl_main()의 _dl_unload_cache()에서 사용한다.

```
316 void
317 _dl_unload_cache (void)
318 {
319   if (cache != NULL && cache != (struct cache_file *) -1)
320     {
321       __munmap (cache, cachesize);
322       cache = NULL;
323     }
324 }
```

glibc-2.26/elf/dl-cache.c에 위치하고 있다.
`[00007f498cf2fbd5] arch_prctl(ARCH_SET_FS, 0x7f498d141700) = 0` 이 부분은 `glibc2.26` 내부의 `dl_main()`내부의 `init_tls()`의 `TLS_INIT_TP()`에 매크로로 구현되어 있다.

```
137 # define TLS_INIT_TP(thrdescr) \
138   ({ void *_thrdescr = (thrdescr);                        \
139      tcbhead_t *_head = _thrdescr;                        \
140      int _result;                                    \
141                                                      \
142      _head->tcb = _thrdescr;                              \
143      /* For now the thread descriptor is at the same address.  */         \
144      _head->self = _thrdescr;                             \
145                                                      \
146      /* It is a simple syscall to set the %fs value for the thread.  */       \
147      asm volatile ("syscall"                              \
148            : "=a" (_result)                          \
149            : "0" ((unsigned long int) __NR_arch_prctl),           \
150              "D" ((unsigned long int) ARCH_SET_FS),           \
151              "S" (_thrdescr)                          \
152            : "memory", "cc", "r11", "cx");                \
```

```
153                                             \
154     _result ? "cannot set %fs base address for thread-local storage" : 0;     \
155   })
```

인라인 어셈블리로 작성되어 있어서, 아키텍쳐별로 각기 다른 파일로 존재한다.
해당 파일은 x86_64이므로, glibc-2.26/sysdeps/x86_64/nptl/tls.h에 위치한다.
ARCH_SET_FS의 옵션의 경우에는 FS segement register를 64bit base로 변환시킬것을 말한다.
주로 TLS와 관련된 구현부에서 나오는데, 서로의 TLS entry를 덮어쓸 수 있으므로, 직접적으로
호출하지 않는 것을 권한다.