人怜直节生来瘦，自许高材老更刚。

Android so注入(inject)和Hook技术学习（一）

以前对Android so的注入只是通过现有的框架，并没有去研究so注入原理，趁现在有时间正好拿出来研究一下。

首先来看注入流程。Android so的注入流程如下：

attach到远程进程 -> 保存寄存器环境 -> 获取目标程序的mmap, dlopen, dlsym, dlclose 地址 -> 远程调用mmap函数申请内存空间用来保存参数信息 -> 向远程进程内存空间写入加载模块名和调用函数->远程调用dlopen函数加载so文件 -> 远程调用dlsym函数获取目标函数地址->使用ptrace_call远程调用被注入模块的函数 -> 调用 dlclose 卸载so文件 -> 恢复寄存器环境 -> 从远程进程detach(进程暂停->ptrace函数调用，其他函数远程调用->进程恢复)

下面我们通过代码来实现这个流程。首先创建目录及文件：

jni
- inject.c
- Android.mk
- Application.mk

在编写代码之前，我们先熟悉一下pt_regs结构体：

```
pt_regs结构的定义：
    struct pt_regs{
        long uregs[18];
    };
    #define ARM_cpsr uregs[16]     存储状态寄存器的值
    #define ARM_pc uregs[15]     存储当前的执行地址
    #define ARM_lr uregs[14]     存储返回地址
    #define ARM_sp uregs[13]     存储当前的栈顶地址
    #define ARM_ip uregs[12]
    #define ARM_fp uregs[11]
    #define ARM_10 uregs[10]
    #define ARM_9 uregs[9]
    #define ARM_8 uregs[8]
    #define ARM_7 uregs[7]
    #define ARM_6 uregs[6]
    #define ARM_5 uregs[5]
    #define ARM_4 uregs[4]
    #define ARM_3 uregs[3]
    #define ARM_2 uregs[2]
    #define ARM_1 uregs[1]
    #define ARM_0 uregs[0]         存储R0寄存器的值，函数调用后的返回值会存储在R0寄存器中
```

在通过ptrace改变远程进程的执行流程之前，需要先读取和保存远程进程的所有寄存器的值，在ARM处理器下，ptrace函数中data参数的regs为pt_regs结构的指针，从远程进程获取的寄存器值将存储到该结构中。在远程进程执行detach操作之前，需要将远程进程的原寄存器的环境恢复，保证远程进程原有的执行流程不被破坏。如果不恢复寄存器的值，则执行detach操作之后会导致远程进程崩溃。

inject.c的代码如下：

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/user.h>
4 #include <asm/ptrace.h>
5 #include <sys/ptrace.h>
6 #include <sys/wait.h>
7 #include <sys/mman.h>
```

```c
#include <dlfcn.h>
#include <dirent.h>
#include <unistd.h>
#include <string.h>
#include <elf.h>
#include <android/log.h>

#if defined(__i386__)
#define pt_regs user_regs_struct
#endif

#define LOG_TAG "INJECT"
#define LOGD(fmt, args...) __android_log_print(ANDROID_LOG_DEBUG, LOG_TAG, fmt, ##args)
#define CPSR_T_MASK (1u << 5)

const char* libc_path = "/system/lib/libc.so";
const char* linker_path = "/system/bin/linker";

/*-------------------------------------------------
*   功能:   向目标进程指定的地址中读取数据
*
*   参数:
*           pid       需要注入的进程pid
*           src       需要读取的目标进程地址
*           buf       需要读取的数据缓冲区
*           size      需要读取的数据长度
*
*   返回值: -1
*-------------------------------------------------*/
int ptrace_readdata(pid_t pid, uint8_t *src, uint8_t *buf, size_t size){
    uint32_t i, j, remain;
    uint8_t *laddr;

    union u{
        long val;
        char chars[sizeof(long)];
    }d;

    j = size/4;
    remain = size%4;
    laddr = buf;

    for(i = 0; i<j; i++){
        //从内存地址src中读取四个字节
        d.val = ptrace(PTRACE_PEEKTEXT, pid, src, 0);
        memcpy(laddr, d.chars, 4);
        src += 4;
        laddr += 4;
    }

    if(remain > 0){
        d.val = ptrace(PTRACE_PEEKTEXT, pid, src, 0);
        memcpy(laddr, d.chars, remain);
    }
    return 0;
}

/*-------------------------------------------------
*   功能:   向目标进程指定的地址中写入数据
*
*   参数:
*           pid       需要注入的进程pid
*           dest      需要写入的目标进程地址
*           data      需要写入的数据缓冲区
*           size      需要写入的数据长度
*
*   返回值: -1
*-------------------------------------------------*/
int ptrace_writedata(pid_t pid, uint8_t *dest, uint8_t *data, size_t size){
    uint32_t i, j, remain;
    uint8_t *laddr;
```

```
79
80      union u{
81          long val;
82          char u_data[sizeof(long)];
83      }d;
84
85      j = size/4;
86      remain = size%4;
87
88      laddr = data;
89
90      //先4字节拷贝
91      for(i = 0; i<j; i++){
92          memcpy(d.u_data, laddr, 4);
93          //往内存地址中写入四个字节,内存地址由dest给出
94          ptrace(PTRACE_POKETEXT, pid, dest, d.val);
95
96          dest += 4;
97          laddr += 4;
98      }
99
100     //最后不足4字节的，单字节拷贝
101     //为了最大程度的保持原栈的数据，需要先把原程序最后四字节读出来
102     //然后把多余的数据remain覆盖掉四字节中前面的数据
103     if(remain > 0){
104         d.val = ptrace(PTRACE_PEEKTEXT, pid, dest, 0);    //从内存地址中读取四个字节，内存地址由dest给出
105         for(i = 0; i<remain; i++){
106             d.u_data[i] = *laddr++;
107         }
108         ptrace(PTRACE_POKETEXT, pid, dest, d.val);
109     }
110     return 0;
111 }
112
113 /*-------------------------------------------------
114 *   功能:   获取指定进程的寄存器信息
115 *
116 *   返回值：失败返回-1
117 *-------------------------------------------------*/
118 int ptrace_getregs(pid_t pid, struct pt_regs *regs){
119     if(ptrace(PTRACE_GETREGS, pid, NULL, regs) < 0){
120         perror("ptrace_getregs: Can not get register values.");
121         return -1;
122     }
123     return 0;
124 }
125
126 /*-------------------------------------------------
127 *   功能:   修改目标进程寄存器的值
128 *
129 *   参数:
130 *           pid         需要注入的进程pid
131 *           pt_regs     需要修改的新寄存器信息
132 *
133 *   返回值：-1
134 *-------------------------------------------------*/
135 int ptrace_setregs(pid_t pid, struct pt_regs *regs){
136     if(ptrace(PTRACE_SETREGS, pid, NULL, regs) < 0){
137         perror("ptrace_setregs:Can not set regsiter values.");
138         return -1;
139     }
140     return 0;
141 }
142
143 /*-------------------------------------------------
144 *   功能:   恢复程序运行
145 *
146 *   参数:
147 *           pid         需要注入的进程pid
148 *
149 *   返回值：-1
```

```
150 *-------------------------------------------------*/
151 int ptrace_continue(pid_t pid){
152     if(ptrace(PTRACE_CONT, pid, NULL, 0) < 0){
153         perror("ptrace_cont");
154         return -1;
155     }
156     return 0;
157 }
158 
159 /*-------------------------------------------------
160 *   功能:   附加进程
161 *
162 *   返回值：失败返回-1
163 *-------------------------------------------------*/
164 int ptrace_attach(pid_t pid){
165     if(ptrace(PTRACE_ATTACH, pid, NULL, 0) < 0){
166         perror("ptrace_attach");
167         return -1;
168     }
169     return 0;
170 }
171 
172 // 释放对目标进程的附加调试
173 int ptrace_detach(pid_t pid)
174 {
175     if (ptrace(PTRACE_DETACH, pid, NULL, 0) < 0) {
176         perror("ptrace_detach");
177         return -1;
178     }
179 
180     return 0;
181 }
182 /*-------------------------------------------------
183 *   功能:   获取进程中指定模块的首地址
184 *    原理:   通过遍历/proc/pid/maps文件，来找到目的module_name的内存映射起始地址。
185 *    由于内存地址的表达方式是startAddrxxxxxxx-endAddrxxxxxxx的，所以通过使用strtok(line,"-")来分割字符串获取地址
186 *    如果pid = -1,表示获取本地进程的某个模块的地址，否则就是pid进程的某个模块的地址
187 *   参数:
188 *           pid             需要注入的进程pid，如果为0则获取自身进程
189 *           module_name         需要获取模块路径
190 *
191 *   返回值：失败返回NULL，成功返回addr
192 *-------------------------------------------------*/
193 void *get_module_base(pid_t pid, const char* module_name)
194 {
195     FILE* fp;
196     long addr = 0;
197     char* pch;
198     char filename[32];
199     char line[1024];
200 
201     if(pid < 0){
202         snprintf(filename, sizeof(filename), "/proc/self/maps");
203     }else{
204         snprintf(filename, sizeof(filename), "/proc/%d/maps", pid);
205     }
206 
207     fp = fopen(filename, "r");
208 
209     if(fp != NULL){
210         while(fgets(line, sizeof(line), fp)){
211             if(strstr(line, module_name)){
212                 pch = strtok(line, "-");
213                 //将参数pch字符串根据参数base(表示进制)来转换成无符号的长整型数
214                 addr = strtoul(pch, NULL, 16);
215                 if(addr == 0x8000)
216                     addr = 0;
217                 break;
218             }
219         }
220         fclose(fp);
```

```
221     }
222     return (void*)addr;
223 }
224 
225 
226 /*-------------------------------------------------
227 *   功能:   获取目标进程中函数指针
228 *
229 *   参数:
230 *           target_pid          需要注入的进程pid
231 *           module_name            需要获取的函数所在的lib库路径
232 *           local_addr           需要获取的函数所在当前进程内存中的地址
233 *
234 *           目标进程中函数指针 = 目标进程模块基址 - 自身进程模块基址 + 内存中的地址
235 *
236 *   返回值：失败返回NULL，成功返回ret_addr
237 *-------------------------------------------------*/
238 void* get_remote_addr(pid_t target_pid, const char* module_name, void* local_addr){
239     void* local_handle, *remote_handle;
240     //获取本地某个模块的起始地址
241     local_handle = get_module_base(-1, module_name);
242     //获取远程pid的某个模块的起始地址
243     remote_handle = get_module_base(target_pid, module_name);
244 
245     LOGD("[+]get remote address: local[%x], remote[%x]\n", local_handle, remote_handle);
246 
247     //local_addr - local_handle的值为指定函数(如mmap)在该模块中的偏移量，然后再加上remote_handle，结果就为指定函数在目标进程的虚拟地址
248     void* ret_addr = (void*)((uint32_t)local_addr - (uint32_t)local_handle) + (uint32_t)remote_handle;
249     return ret_addr;
250 }
251 
252 /*-------------------------------------------------
253 *   功能:   通过进程的名称获取对应的进程pid
254 *   原理:   通过遍历/proc目录下的所有子目录，获取这些子目录的目录名(一般就是进程的进程号pid)。
255 *             获取子目录名后，就组合成/proc/pid/cmdline文件名，然后依次打开这些文件，cmdline文件
256 *             里面存放的就是进程名，通过这样就可以获取进程的pid了
257 *   返回值：未找到返回-1
258 *-------------------------------------------------*/
259 int find_pid_of(const char* process_name){
260     int id;
261     pid_t pid = -1;
262     DIR* dir;
263     FILE* fp;
264     char filename[32];
265     char cmdline[296];
266 
267     struct dirent* entry;
268 
269     if(process_name == NULL){
270         return -1;
271     }
272 
273     dir = opendir("/proc");
274     if(dir == NULL){
275         return -1;
276     }
277 
278     while((entry = readdir(dir)) != NULL){
279         id = atoi(entry->d_name);
280         if(id != 0){
281             sprintf(filename, "/proc/%d/cmdline", id);
282             fp = fopen(filename, "r");
283             if(fp){
284                 fgets(cmdline, sizeof(cmdline), fp);
285                 fclose(fp);// 释放对目标进程的附加调试
286 
287                 if(strcmp(process_name, cmdline) == 0){
288                     pid = id;
289                     break;
290                 }
```

```
291             }
292         }
293     }
294     closedir(dir);
295     return pid;
296 }
297
298 long ptrace_retval(struct pt_regs* regs){
299     return regs->ARM_r0;
300 }
301
302 long ptrace_ip(struct pt_regs* regs){
303     return regs->ARM_pc;
304 }
305
306 /*-------------------------------------------------
307 *   功能:   调用远程函数指针
308 *    原理:     1，将要执行的指令写入寄存器中，指令长度大于4个long的话，需要将剩余的指令通过ptrace_writedata函数写入栈中;
309 *            2，使用ptrace_continue函数运行目的进程，直到目的进程返回状态值0xb7f（对该值的分析见后面红字）；
310 *            3，函数执行完之后，目标进程挂起，使用ptrace_getregs函数获取当前的所有寄存器值，方便后面使用ptrace_retval函数
    获取函数的返回值。
311 *   参数:
312 *           pid             需要注入的进程pid
313 *           addr             调用的函数指针地址
314 *           params          调用的参数
315 *           num_params      调用的参数个数
316 *           regs            远程进程寄存器信息(ARM前4个参数由r0 ~ r3传递)
317 *
318 *   返回值：失败返回-1
319 *-------------------------------------------------*/
320 int ptrace_call(pid_t pid, uint32_t addr, long* params, uint32_t num_params, struct pt_regs* regs){
321     uint32_t i;
322     for(i = 0; i<num_params && i < 4; i++){
323         regs->uregs[i] = params[i];
324     }
325
326     if(i < num_params){
327         regs->ARM_sp -= (num_params - i) * sizeof(long);
328         ptrace_writedata(pid, (void*)regs->ARM_sp, (uint8_t*)&params[i], (num_params - i)*sizeof(long));
329     }
330     //将PC寄存器值设为目标函数的地址
331     regs->ARM_pc = addr;
332     ////指令集判断
333     if(regs->ARM_pc & 1){
334         /* thumb */
335         regs->ARM_pc &= (~1u);
336         regs->ARM_cpsr |= CPSR_T_MASK;
337     }else{
338          /* arm */
339         regs->ARM_cpsr &= ~CPSR_T_MASK;
340     }
341     ///设置子程序的返回地址为空，以便函数执行完后，返回到null地址，产生SIGSEGV错误
342     regs->ARM_lr = 0;
343
344     //将修改后的regs写入寄存器中，然后调用ptrace_continue来执行我们指定的代码
345     if(ptrace_setregs(pid, regs) == -1 || ptrace_continue(pid) == -1){
346         printf("error.\n");
347         return -1;
348     }
349
350     int stat = 0;
351     /* WUNTRACED告诉waitpid，如果子进程进入暂停状态，那么就立即返回。如果是被ptrace的子进程，那么即使不提供WUNTRACED参数，
    也会在子进程进入暂停状态的时候立即返回。
352     对于使用ptrace_cont运行的子进程，它会在3种情况下进入暂停状态：①下一次系统调用；②子进程退出；③子进程的执行发生错误。这里
    的0xb7f就表示子进程进入了暂停状态，
353     且发送的错误信号为11(SIGSEGV)，它表示试图访问未分配给自己的内存，或试图往没有写权限的内存地址写数据。那么什么时候会发生这
    种错误呢？显然，当子进程执行完注入的
354     函数后，由于我们在前面设置了regs->ARM_lr = 0，它就会返回到0地址处继续执行，这样就会产生SIGSEGV了！
355     */
356     waitpid(pid, &stat, WUNTRACED);
357     /*stat的值：高2字节用于表示导致子进程的退出或暂停状态信号值，低2字节表示子进程是退出(0x0)还是暂停(0x7f)状态。
```

```
358     0xb7f就表示子进程为暂停状态，导致它暂停的信号量为11即sigsegv错误。*/
359     while(stat != 0xb7f){
360         if(ptrace_continue(pid) == -1){
361             printf("error.\n");
362             return -1;
363         }
364         waitpid(pid, &stat, WUNTRACED);
365     }
366     return 0;
367 }
368
369 /*-------------------------------------------------
370 *   功能：   调用远程函数指针
371 *
372 *   参数：
373 *           pid             需要注入的进程pid
374 *           func_name       调用的函数名称，此参数仅作Debug输出用
375 *           func_addr       调用的函数指针地址
376 *           param           调用的参数
377 *           param_num       调用的参数个数
378 *           regs            远程进程寄存器信息(ARM前4个参数由r0 ~ r3传递)
379 *
380 *   返回值：失败返回-1
381 *-------------------------------------------------*/
382 int ptrace_call_wrapper(pid_t target_pid, const char* func_name, void* func_addr, long* param, int param_num
, struct pt_regs* regs){
383     LOGD("[+]Calling %s in target process.\n", func_name);
384     if(ptrace_call(target_pid, (uint32_t)func_addr, param, param_num, regs) == -1)
385         return -1;
386     if(ptrace_getregs(target_pid, regs) == -1){
387         return -1;
388     }
389     LOGD("[+] Target process returned from %s, return value = %x, pc = %x \n", func_name, ptrace_retval(regs
), ptrace_ip(regs));
390     return 0;
391 }
392
393 /*-------------------------------------------------
394 *   功能：   远程注入
395 *
396 *   参数：
397 *           target_pid          需要注入的进程Pid
398 *           library_path           需要注入的.so路径
399 *           function_name          .so中导出的函数名
400 *           param                 函数的参数
401 *            param_size            参数大小，以字节为单位
402 *
403 *   返回值：注入失败返回-1
404 *-------------------------------------------------*/
405 int inject_remote_process(pid_t target_pid, const char* library_path, const char* function_name, const char*
param, size_t param_size){
406     int ret = -1;
407     void* mmap_addr, *dlopen_addr, *dlsym_addr, *dlclose_addr, *dlerror_addr;
408     void *local_handle, *remote_handle, *dlhandle;
409     uint8_t *map_base = 0;
410     uint8_t *dlopen_param1_ptr, *dlsym_param2_ptr, *saved_r0_pc_ptr, *inject_param_ptr, *remote_code_ptr, *l
ocal_code_ptr;
411
412     struct pt_regs regs, original_regs;
413     extern uint32_t _dlopen_addr_s, _dlopen_param1_s, _dlopen_param2_s, _dlsym_addr_s, _dlsym_param2_s, _dlc
lose_addr_s, _inject_start_s, _inject_end_s, _inject_function_param_s, _saved_cpsr_s, _saved_r0_pc_s;
414
415     uint32_t code_length;
416     long parameters[10];
417
418     LOGD("[+] Injecting process: %d\n", target_pid);
419
420     //①ATTATCH，指定目标进程，开始调试
421     if(ptrace_attach(target_pid) == -1){
422         goto exit;
423     }
```

```
424
425     //②GETREGS，获取目标进程的寄存器，保存现场
426     if(ptrace_getregs(target_pid, &regs) == -1)
427         goto exit;
428
429     //保存原始寄存器
430     memcpy(&original_regs, &regs, sizeof(regs));
431
432     //③通过get_remote_addr函数获取目标进程的mmap函数的地址，以便为libxxx.so分配内存
433     //由于mmap函数在libc.so库中，为了将libxxx.so加载到目标进程中，就需要使用目标进程的mmap函数，所以需要查找到libc.so库在目标进程的起始地址。
434     mmap_addr = get_remote_addr(target_pid, libc_path, (void*)mmap);    //libc_path = "/system/lib/libc.so"
435     LOGD("[+] Remote mmap address: %x\n", mmap_addr);
436
437     parameters[0] = 0;    // 设置为NULL表示让系统自动选择分配内存的地址
438     parameters[1] = 0x4000;    // 映射内存的大小
439     parameters[2] = PROT_READ | PROT_WRITE |PROT_EXEC;    // 表示映射内存区域可读可写可执行
440     parameters[3] = MAP_ANONYMOUS | MAP_PRIVATE;    // 建立匿名映射
441     parameters[4] = 0;    //若需要映射文件到内存中，则为文件的fd
442     parameters[5] = 0;    //文件映射偏移量
443
444     //④通过ptrace_call_wrapper调用mmap函数，在目标进程中为libxxx.so分配内存
445     if(ptrace_call_wrapper(target_pid, "mmap", mmap_addr, parameters, 6, &regs) == -1)
446         goto exit;
447     //⑤从寄存器中获取mmap函数的返回值，即申请的内存首地址：
448     map_base = ptrace_retval(&regs);
449
450     //⑥依次获取linker中dlopen、dlsym、dlclose、dlerror函数的地址
451     dlopen_addr = get_remote_addr(target_pid, linker_path, (void*)dlopen);
452     dlsym_addr = get_remote_addr(target_pid, linker_path, (void*)dlsym);
453     dlclose_addr = get_remote_addr(target_pid, linker_path, (void*)dlclose);
454     dlerror_addr = get_remote_addr(target_pid, linker_path, (void*)dlerror);
455
456     LOGD("[+] Get imports: dlopen: %x, dlsym: %x, dlclose: %x, dlerror: %x\n", dlopen_addr, dlsym_addr, dlclose_addr, dlerror_addr);
457
458     printf("library path = %s\n", library_path);
459     //⑦调用dlopen函数
460     //(1)将要注入的so名写入前面mmap出来的内存
461     ptrace_writedata(target_pid, map_base, library_path, strlen(library_path) + 1);
462
463     parameters[0] = map_base;
464     parameters[1] = RTLD_NOW | RTLD_GLOBAL;
465
466     //(2)执行dlopen
467     if(ptrace_call_wrapper(target_pid, "dlopen", dlopen_addr, parameters, 2, &regs) == -1){
468         goto exit;
469     }
470     //(3)取得dlopen的返回值，存放在sohandle变量中
471     void* sohandle = ptrace_retval(&regs);
472
473     //⑧调用dlsym函数
474     //为functionname另找一块区域
475     #define FUNCTION_NAME_ADDR_OFFSET 0X100
476     ptrace_writedata(target_pid, map_base + FUNCTION_NAME_ADDR_OFFSET, function_name, strlen(function_name) + 1);
477     parameters[0] = sohandle;
478     parameters[1] = map_base + FUNCTION_NAME_ADDR_OFFSET;
479
480     //调用dlsym
481     if(ptrace_call_wrapper(target_pid, "dlsym", dlsym_addr, parameters, 2, &regs) == -1)
482         goto exit;
483     void* hook_entry_addr = ptrace_retval(&regs);
484     LOGD("hooke_entry_addr = %p\n", hook_entry_addr);
485
486     //⑨调用被注入函数hook_entry
487     #define FUNCTION_PARAM_ADDR_OFFSET 0X200
488     ptrace_writedata(target_pid, map_base + FUNCTION_PARAM_ADDR_OFFSET, parameters, strlen(parameters) + 1);
489     parameters[0] = map_base + FUNCTION_PARAM_ADDR_OFFSET;
490
491     if(ptrace_call_wrapper(target_pid, "hook_entry", hook_entry_addr, parameters, 1, &regs) == -1)
```

```
492         goto exit;
493     //⑩调用dlclose关闭lib
494     printf("Press enter to dlclose and detach.\n");
495     getchar();
496     parameters[0] = sohandle;
497 
498     if(ptrace_call_wrapper(target_pid, "dlclose", dlclose, parameters, 1, &regs) == -1)
499         goto exit;
500 
501     //⑪恢复现场并退出ptrace
502     ptrace_setregs(target_pid, &original_regs);
503     ptrace_detach(target_pid);
504     ret = 0;
505 
506 exit:
507     return ret;
508 }
509 
510 int main(int argc, char** argv) {
511     pid_t target_pid;
512     target_pid = find_pid_of("com.bbk.appstore");
513     if (-1 == target_pid) {
514         printf("Can't find the process\n");
515         return -1;
516     }
517     //target_pid = find_pid_of("/data/test");
518     inject_remote_process(target_pid, "/data/local/tmp/libentry.so", "hook_entry",  "Fuck you!", strlen("Fuc
k you!"));
519     return 0;
520 }
```

上述代码中，我们要hook的进程名为"com.bbk.appstore"，我们要将"libentry.so"注入到该进程中去。

Android.mk内容为：

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE := inject
LOCAL_SRC_FILES := inject.c

#shellcode.s

LOCAL_LDLIBS += -L$(SYSROOT)/usr/lib -llog

#LOCAL_FORCE_STATIC_EXECUTABLE := true

include $(BUILD_EXECUTABLE)
```

Application.mk内容为：

```
# 编译生成的模块文件运行支持的平台
APP_ABI := armeabi-v7a
# 编译生成模块运行支持的Andorid版本
APP_PLATFORM := android-19
```

在jni目录下运行nkd-build编译成生arm平台下的可执行文件：

```
ndk-build NDK_PROJECT_PATH=. APP_BUILD_SCRIPT=./Android.mk NDK_APPLICATION_MK=./Application.mk
```

再来生成要注入的so，创建目录及文件：

jni
    entry.c
    Android.mk
    Application.mk

entry.c的代码为：

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <android/log.h>
#include <elf.h>
#include <fcntl.h>

#define LOG_TAG "DEBUG"
#define LOGD(fmt, args...) __android_log_print(ANDROID_LOG_DEBUG, LOG_TAG, fmt, ##args)

int hook_entry(char * a){
    LOGD("Hook success, pid = %d\n", getpid());
    LOGD("Hello %s\n", a);
    return 0;
}
```

Android.mk文件：

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_LDLIBS += -L$(SYSROOT)/usr/lib -llog
#LOCAL_ARM_MODE := arm
LOCAL_MODULE    := entry
LOCAL_SRC_FILES := entry.c
include $(BUILD_SHARED_LIBRARY)
```

Application.mk文件内容跟上面一样。同样将entry.c进行编译。然后将得到inject和libentry.so push到/data/local/tmp目录下，执行：

```
root@hammerhead:/data/local/tmp # ./inject
library path = /data/local/tmp/libentry.so
Press enter to dlclose and detach.
```

通过"/proc/pid/maps"查看被注入进程（"com.bbk.appstore"）的mmap，可以看到我们的so已经被加载了：

```
76592000-76593000 rw-p 00000000 00:00 0          [anon:libc_malloc]
76593000-76595000 r-xp 00000000 b3:1c 627134     /data/local/tmp/libentry.so
76595000-76596000 r--p 00001000 b3:1c 627134     /data/local/tmp/libentry.so
76596000-76597000 rw-p 00002000 b3:1c 627134     /data/local/tmp/libentry.so
76597000-76598000 rw-p 00000000 00:00 0          [anon:libc_malloc]
7659a000-765a7000 rw-p 00000000 00:00 0          [anon:libc_malloc]
765a7000-765a8000 r--p 00000000 00:00 0
```

通过"adb logcat -s INJECT"命令打印出log：

```
root@ubuntu:~# adb logcat -s INJECT
--------- beginning of /dev/log/system
--------- beginning of /dev/log/main
D/INJECT  ( 5452): [+] Injecting process: 4530
D/INJECT  ( 5554): [+] Injecting process: 4530
D/INJECT  ( 5554): [+]get remote address: local[b6f78000], remote[400d0000]
D/INJECT  ( 5554): [+] Remote mmap address: 400e2d31
D/INJECT  ( 5554): [+]Calling mmap in target process.
D/INJECT  ( 5554): [+] Target process returned from mmap, return value = 7670e000, pc = 0
D/INJECT  ( 5554): [+]get remote address: local[b6fdb000], remote[400b5000]
D/INJECT  ( 5554): [+]get remote address: local[b6fdb000], remote[400b5000]
D/INJECT  ( 5554): [+]get remote address: local[b6fdb000], remote[400b5000]
D/INJECT  ( 5554): [+]get remote address: local[b6fdb000], remote[400b5000]
D/INJECT  ( 5554): [+] Get imports: dlopen: 400b5f4d, dlsym: 400b5e9d, dlclose: 400b5e19, dlerror: 400b5dc9
D/INJECT  ( 5554): [+]Calling dlopen in target process.
D/INJECT  ( 5554): [+] Target process returned from dlopen, return value = 764bda48, pc = 0
D/INJECT  ( 5554): [+]Calling dlsym in target process.
D/INJECT  ( 5554): [+] Target process returned from dlsym, return value = 78289c59, pc = 0
D/INJECT  ( 5554): hooke_entry_addr = 0x78289c59
D/INJECT  ( 5554): [+]Calling hook_entry in target process.
D/INJECT  ( 5554): [+] Target process returned from hook_entry, return value = 0, pc = 0
```

这就说明我们的注入成功了。

参考资料:

https://blog.csdn.net/qq1084283172/article/details/53942648
https://melonwxd.github.io/2017/12/01/inject-3-hook/
https://www.cnblogs.com/wanyuanchun/p/4020756.html

分类: Android底层

bamb00
关注 - 1
粉丝 - 167
+加关注

« 上一篇: 虫洞漏洞挖掘研究
» 下一篇: Android so注入(inject)和Hook技术学习（二）——Got表hook之导入表hook

posted @ 2018-07-11 21:57 bamb00 阅读(688) 评论(1) 编辑 收藏