Ptrace injection of mobile game plug-in basics

yong night (/u/15648) / 2019-06-10 06:01:00 / views 12004

[TOC]

introduction

This article specifically implements how to inject ptrace from the code to prepare for the implementation of injecting plug-in modules into the game process to crack the mobile game 2048

Technical overview

Mainly through the function of the system call function ptrace:

1. Inject the module into the remote process through shellcode
2. Use ptrace to remotely call dlopen/dlsym to inject the dynamic link library into the remote process and perform corresponding operations.

The following is also the main implementation of dlopen/dlsym for so library injection and function call

Code

Header file : declare function

```
/**********************************
 *  FileName:   ptraceInject.h
 *  Decription: ptrace注入
 * ******************************/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/* Function 1: Inject the module into the remote process by calling dlopen/dlsym remotely by ptrace*/
int  inject_remote_process ( pid_t  pid ,  char  * LibPath ,  char  * FunctionName ,  long  *
FuncParameter ,  long  NumParameter );

/* Function 2: Inject the module into the remote process through shellcode*/
int  inject_remote_process_shellcode ( pid_t  pid ,  char  * LibPath ,  char  * FunctionName ,  long  *
FuncParameter ,  long  NumParameter );
```

Log tools used for debugging : `PrintLog.h`

```
#ifndef _ANDROID_LOG_PRINT_H_
#define _ANDROID_LOG_PRINT_H_

#define  MAX_PATH 0x100

#include <android/log.h>
//If you don’t want to print the log, you can comment this line of macro definition
#define IS_DEBUG
//If the macro defines IS_DEBUG, then the following macro definitions will be made for the following log
printing functions
#ifdef IS_DEBUG

#define LOG_TAG ("INJECT")

#define LOGV(...) ((void)__android_log_print(ANDROID_LOG_VERBOSE, LOG_TAG, __VA_ARGS__))

#define LOGD(...) ((void)__android_log_print(ANDROID_LOG_DEBUG  , LOG_TAG, __VA_ARGS__))

#define LOGI(...) ((void)__android_log_print(ANDROID_LOG_INFO   , LOG_TAG, __VA_ARGS__))

#define LOGW(...) ((void)__android_log_print(ANDROID_LOG_WARN   , LOG_TAG, __VA_ARGS__))

#define LOGE(...) ((void)__android_log_print(ANDROID_LOG_ERROR  , LOG_TAG, __VA_ARGS__))

#else

#define LOGV(LOG_TAG, ...) NULL

#define LOGD(LOG_TAG, ...) NULL

#define LOGI(LOG_TAG, ...) NULL

#define LOGW(LOG_TAG, ...) NULL

#define LOGE(LOG_TAG, ...) NULL

#endif

#endif
```

Source code : ptraceInject.c

```
/*******************************
 *  FileName: ptraceInject.c
 *  Description:       ptrace注入
 * *****************************/


#include <stdio.h>
#include <stdlib.h>
#include <sys/user.h>
#include <asm/ptrace.h>
#include <sys/ptrace.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <dlfcn.h>
#include <dirent.h>
#include <unistd.h>
#include <string.h>
#include <elf.h>
#include <PrintLog.h>
#include <ptraceInject.h>

#define CPSR_T_MASK     ( 1u << 5 )

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


/***************************
* Description: Use ptrace Attach to attach to the specified process, send a SIGSTOP signal to the
specified process to stop it and Keep track of it.
* But the traced process (tracee) will not necessarily stop, because attach and pass SIGSTOP at the same
time may lose SIGSTOP.
* So waitpid(2) is needed to wait for the tracked process to be stopped
* Input: pid represents the ID of the remote process
* Output: none
* Return: return 0 means attach is successful, return -1 means failure
* Others: none
* **** ********************/
int  ptrace_attach ( pid_t  pid )
{
    int  status  =  0 ;
    if  ( ptrace ( PTRACE_ATTACH ,  pid ,  NULL ,  NULL) < 0)
    {
        LOGD("ptrace attach error, pid:%d", pid);
        return -1;
    }

    LOGD("attach process pid:%d", pid);
    waitpid(pid, &status , WUNTRACED);
    return 0;
}

/*************************************************
* Description: Use ptrace detach to specify the process, after completing the trace operation of the
specified process, use this parameter to unattach it
* Input: pid represents the ID of the remote process
* Output: none
* Return: return 0 to indicate detach success, return -1 Indicates failure
* Others: None
* ******************************************* ****/
int  ptrace_detach ( pid_t  pid )
{
    if  ( ptrace ( PTRACE_DETACH ,  pid ,  NULL ,  0 )  <  0 )
    {
        LOGD ( "detach process error, pid:%d" ,  pid);
        return -1;
    }
    LOGD("detach process pid:%d", pid);
    return 0;
}

/*************************************************
* Description: ptrace makes the remote process continue to run
* Input: pid represents the ID of the remote process
* Input: pid represents the ID of the remote process
* Output: none
* Return: return 0 to indicate success, return -1 to indicate failure
* Others: none
* ********* **************************************/
int  ptrace_continue ( pid_t  pid )
{
    if  ( ptrace ( PTRACE_CONT ,  pid ,  NULL ,  NULL )  <  0 )
    {
        LOGD ( "ptrace cont error, pid:%d" ,  pid );
        return -1;
    }
    return 0;
}

/************************************************
* Description: Use ptrace to obtain the register value of the remote process
* Input: pid represents the ID of the remote process, regs is the pt_regs structure, and the register
value is stored
* Output: None
* Return: Return 0 to indicate success in obtaining the register, return -1 to indicate failure
* Others: None
* ********************************************** */
int  ptrace_getregs ( pid_t  pid ,  struct  pt_regs  * regs )
{
    if  ( ptrace ( PTRACE_GETREGS ,  pid ,  NULL ,  regs )  <  0 )
    {
        LOGD ("Get Regs error, pid:%d", pid);
        return -1;
    }
    return 0;
}

/************************************************
* Description: Use ptrace to set the register value of the remote process
* Input: pid represents the ID of the remote process, regs is the pt_regs structure, and stores the
register value that needs to be modified
* Output: None
* Return: Return 0 to indicate success in setting the register, return -1 to indicate Failed
* ************************************************/
int  ptrace_setregs ( pid_t  pid ,  struct  pt_regs  * regs )
{
    if  ( ptrace ( PTRACE_SETREGS ,  pid ,  NULL ,  regs )  <  0 )
    {
        LOGD ("Set Regs error, pid:%d", pid);
        return -1;
    }
    return 0;
}

/************************************************
* Description: Get the return value, the return value in the ARM processor is stored in the ARM_r0
register
* Input: regs stores the current register value of the remote process
* Return: The value of the r0 register is returned under the ARM processor
* ******** **************************************/
long  ptrace_getret ( struct  pt_regs  *  regs )
{
    return  regs -> ARM_r0 ;
}

/************************************************
* Description: get the address of the currently executing code stored in ARM_pc under ARM processor
* Input: regs remote storage course of the current register value
* return: return pc register values in the ARM processor
* ******** *************************************/
long  ptrace_getpc ( struct  pt_regs  *  regs )
{
    return  regs -> ARM_pc ;
}

/************************************************
* Description: Use ptrace to read data from the memory of the remote process
* Input: pid represents the ID of the remote process, pSrcBuf represents the memory address of the data
read from the remote process
* pDestBuf represents the
address used to store the read data, size represents read Get
address used to store the read data, size represents read Get
the size of the data * Return: Return 0 to indicate that the data is successfully read. * Other: The *_t
type here is an alias of some basic types defined by typedef, which is used for cross-platform. For
example
* uint8_t means unsigned 8-bit, which is unsigned char type
* ********************************* ************/
int  ptrace_readdata ( pid_t  pid ,  uint8_t  * pSrcBuf ,  uint8_t  * pDestBuf ,  uint32_t  size )
{
    uint32_t  nReadCount = 0;
    uint32_t nRemainCount = 0;
    uint8_t *pCurSrcBuf = pSrcBuf;
    uint8_t *pCurDestBuf = pDestBuf;
    long lTmpBuf = 0;
    uint32_t i = 0;

    //Read 4 bytes of data each time
    nReadCount  =  size  /  sizeof ( long );
    nRemainCount  =  size  %  sizeof ( long );
    for  ( i  =  0 ;  i  <  nReadCount ;  i ++ )
    {
        lTmpBuf  =  ptrace ( PTRACE_PEEKTEXT ,  pid ,  pCurSrcBuf ,  0 );
        memcpy ( pCurDestBuf ,  ( char  * )( &lTmpBuf ),  sizeof ( long ));
        pCurSrcBuf  +=  sizeof ( long );
        pCurDestBuf  +=  sizeof ( long );
    }
    //Call
    if  (  nRemainCount  >  0  )
    {
        lTmpBuf  =  ptrace ( PTRACE_PEEKTEXT ,  pid ,  pCurSrcBuf ,  0 );
        memcpy ( pCurDestBuf ,  ( char  * )( & lTmpBuf ), nRemainCount);
    }
    return 0;
}

/**************************************************
* Description: Use ptrace to write data into the remote process space
* Input: pid represents the ID of the remote process, pWriteAddr represents the memory address of the
write data to the remote process
* pWriteData is used to store the address of the written data, size represents the write The size of the
data
* Return: Return 0 to indicate success in writing data, return -1 to indicate failure in writing data
* ************************* *******************/
int  ptrace_writedata ( pid_t  pid ,  uint8_t  * pWriteAddr ,  uint8_t  * pWriteData ,  uint32_t  size )
{
    uint32_t  nWriteCount  =  0 ;
    uint32_t  nRemainCount  =  0;
    uint8_t *pCurSrcBuf = pWriteData;
    uint8_t *pCurDestBuf = pWriteAddr;
    long lTmpBuf = 0;
    uint32_t i = 0;

    nWriteCount = size / sizeof(long);
    nRemainCount = size % sizeof(long);

    //Data is written to the remote process memory space in units of sizeof(long) bytes
    for  ( i  =  0 ;  i  <  nWriteCount ;  i  ++ )
    {
        memcpy (( void  * )( & lTmpBuf ),  pCurSrcBuf ,  sizeof ( long ));
        if  ( ptrace ( PTRACE_POKETEXT ,  pid ,  pCurDestBuf ,  lTmpBuf )  <  0 )
        {
            LOGD ( "Write Remote Memory error, MemoryAddr:0x%lx" , (long)pCurDestBuf);
            return -1;
        }
        pCurSrcBuf += sizeof(long);
        pCurDestBuf += sizeof(long);
    }
    if (nRemainCount > 0)
    {
        //lTmpBuf = ptrace(PTRACE_PEEKTEXT, pid, pCurDestBuf, NULL);
        memcpy((void *)(&lTmpBuf), pCurSrcBuf, nRemainCount);
        if (ptrace(PTRACE_POKETEXT, pid, pCurDestBuf, lTmpBuf) < 0)
        {
            LOGD("Write Remote Memory error, MemoryAddr:0x%lx", (long)pCurDestBuf);
            return -1;
        }
    }
    return 0;
    return 0;
}
```

```
/*************************************************
* Description: Use ptrace remote call function
* Input: pid represents the ID of the remote process, ExecuteAddr is the address of the remote process function
* parameters is the address of the function parameter, regs is the register environment before the remote process call function
* Return: return 0 for call Function succeeds, return -1 means failure
* **************************************** *****/
int  ptrace_call ( pid_t  pid ,  uint32_t  ExecuteAddr ,  long  * parameters ,  long  num_params , struct  pt_regs *  regs )
{
    int  i = 0 ;
    // ARM processor, transfer function parameters into the first four parameters r0-r3, remaining parameters onto the stack
    for ( I = 0 ;  I < num_params  &&  I < . 4 ;  I ++ )
    {
       regs - > uregs [ i ]  =  parameters [ i ];
    }
    if ( i  <  num_params )
    {
        regs -> ARM_sp  -=  ( num_params  -  i )  *  sizeof ( long );
        if (ptrace_writedata(pid, (void *)regs->ARM_sp, (uint8_t *)&parameters[i], (num_params - i) * sizeof(long))  == -1)
        {
            return -1;
        }
    }

    //Modify the program counter
    regs -> ARM_pc  =  ExecuteAddr ;

    //Judgment instruction set
    // Similar to the BX jump instruction, it is judged whether the address bit [0] of the jump is 1, if it is 1, the flag T of the CPST register is set and interpreted as Thumb code
    if  ( regs -> ARM_pc  &  1 )
    {
        /*Thumb*/
        regs -> ARM_pc  &=  ( ~ 1u );
        regs -> ARM_cpsr  |=  CPSR_T_MASK ;
    }
    else
    {
        /* ARM*/
        regs -> ARM_cpsr  &=  ~ CPSR_T_MASK ;
    }

    regs->ARM_lr = 0;

    //After setting the registers, start the process
    if  ( ptrace_setregs ( pid ,  regs )  ==  - 1  ||  ptrace_continue ( pid )  ==  - 1 )
    {
        LOGD ( "ptrace set regs or continue error, pid:%d" ,  pid );
        return  - 1 ;
    }

    //For the process running by ptrace_continue, he will enter the suspended state in three cases: 1. The next system call 2. The child process is abnormal 3. The child process exits
    // The parameter WUNTRACED means that when the process enters the suspended state, it returns immediately
    //Set the lr register that stores the return address to 0, an error will occur when the execution returns, and the child process is suspended
    int  stat  =  0 ;
    waitpid ( pid ,  & stat ,  WUNTRACED );
    LOGD ( "ptrace call ret status is% d \n " ,  stat );
    //0xb7f indicates that the child process enters the suspended state
    while  ( stat  !=  0xb7f )
    {
        if  ( ptrace_continue ( pid )  ==  - 1 )
        {
           LOGD ( "ptrace call error" );
           return  - 1 ;
        }
        waitpid ( pid ,  & stat ,  WUNTRACED );
    }
    // Get the register value of the remote process to facilitate the return value
    if  ( ptrace_getregs ( pid ,  regs )  ==  - 1 )
    {
        LOGD ( "After call getregs error" );
        return  - 1 ;
    }
    return  0 ;
}

/*************************************************
* Description: Search for the base address of the corresponding module in the specified process
* Input: pid indicates the ID of the remote process, if -1 indicates the own process, ModuleName
indicates the name of the module to be searched
* Return: Return 0 indicates failure to obtain the module base address , Return non-zero as the base
address of the module to be searched
* ************************************** ********/
void *  GetModuleBaseAddr ( pid_t  pid ,  const  char *  ModuleName )
{
    char  szFileName [ 50 ]  =  { 0 };
    FILE  * fp  =  NULL ;
    char  szMapFileLine [ 1024 ] = {0};
    char *ModulePath, *MapFileLineItem;
    long ModuleBaseAddr = 0;

    // Read "/proc/pid/maps" to get the module loaded by the process
    if  ( pid  <  0 )
    {
        snprintf ( szFileName ,  sizeof ( szFileName ),  "/proc/self/maps" );
    }
    else
    {
        snprintf ( szFileName ,  sizeof ( szFileName ),  "/proc/%d/maps" ,  pid );
    }

    fp = fopen(szFileName, "r");
    if (fp != NULL)
    {
        while (fgets(szMapFileLine, sizeof(szMapFileLine), fp))
        {
            if (strstr(szMapFileLine, ModuleName))
            {
                MapFileLineItem = strtok(szMapFileLine, " \t");
                char *Addr = strtok(szMapFileLine, "-");
                ModuleBaseAddr = strtoul(Addr, NULL, 16 );

                if (ModuleBaseAddr == 0x8000)
                {
                    ModuleBaseAddr = 0;
                }
                break;
            }
        }
        fclose(fp);
    }
    return (void *)ModuleBaseAddr;
}

/*************************************************
* Description: Get the address of the function in the module loaded by both the remote process and this
process
* Input: pid represents the ID of the remote process, ModuleName represents the module name,
LocalFuncAddr represents the address of the function in the local process
* Return: Return the corresponding function in the remote process Address
* *********************************************** /
void *  GetRemoteFuncAddr ( pid_t  pid ,  const  char  * ModuleName ,  void  * LocalFuncAddr )
{
    void  * LocalModuleAddr ,  * RemoteModuleAddr ,  * RemoteFuncAddr ;
    LocalModuleAddr  = GetModuleBaseAddr(-1, ModuleName);
    RemoteModuleAddr = GetModuleBaseAddr(pid, ModuleName);
    RemoteFuncAddr = (void *)((long)LocalFuncAddr - (long)LocalModuleAddr + (long)RemoteModuleAddr);
    return RemoteFuncAddr;
}

/*************************************************
* Directly call dlopen\dlsym remotely to inject the so module into the remote process with ptrace
* Input: pid represents the ID of the remote process, LibPath is the path of the so module injected
remotely, and FunctionName is the function called after the remotely injected module
* FuncParameter Point to the parameter of the remotely called function (if you pass a string, you need to
write the string to the remote process space first), NumParameter is the number of parameters
* Return: return 0 to indicate successful injection, return -1 to indicate failure
* ** ********************************************/
int  inject_remote_process ( pid_t  pid ,  char  * LibPath ,  char  * FunctionName ,  long  *
FuncParameter ,  long  NumParameter )
{
    int  iRet = -1;
    struct pt_regs CurrentRegs, OriginalRegs;
    void *mmap_addr, *dlopen_addr, *dlsym_addr, *dlclose_addr, *dlerror_addr;
    void *RemoteMapMemoryAddr, *RemoteModuleAddr, *RemoteModuleFuncAddr;
    long parameters[6];

    /* 1. Attach to the remote process*/
    if  ( ptrace_attach ( pid )  ==  - 1 )
    {
        return  iRet ;
    }

    /* 2. Get the register value of the remote process and save it, in order to prepare for the program
to resume execution after the module is injected*/
    if  ( ptrace_getregs ( pid ,  & CurrentRegs )  ==  - 1 )
    {
        ptrace_detach ( pid );
        return  iRet ;
    }
    LOGD ( "ARM_r0:0x%lx, ARM_r1:0x%lx, ARM_r2:0x%lx, ARM_r3:0x%lx, ARM_r4:0x%lx, \
          ARM_r5:0x%lx, ARM_r6:0x%lx, ARM_r7:0x %lx, ARM_r8:0x%lx, ARM_r9:0x%lx, \
          ARM_r10:0x%lx, ARM_ip:0x%lx, ARM_sp:0x%lx, ARM_lr:0x%lx, ARM_pc:0x%lx" ,
          CurrentRegs . ARM_r0 ,  CurrentRegs . ARM_r1, CurrentRegs.ARM_r2, CurrentRegs.ARM_r3,
CurrentRegs.ARM_r4,
          CurrentRegs.ARM_r5, CurrentRegs.ARM_r6, CurrentRegs.ARM_r7, CurrentRegs.ARM_r8,
CurrentRegs.ARM_r9,
          CurrentRegs.ARM_r10, CurrentRegs.ARM_ip, CurrentRegs.ARM_sp, CurrentRegs.ARM_lr,
CurrentRegs.ARM_pc);
    memcpy(&OriginalRegs, &CurrentRegs, sizeof(CurrentRegs));

    /* 3. Open up a memory space inside the remote process to store some constant data, and provide
parameter addresses for the remote process to execute function calls
     * Here you need to know, why don't we pass in the constants directly? This is because the value we
are passing now is the value of the memory space of our current injection tool,
     * The corresponding memory address is also our injection tool, and the remote process is not
accessible, so we need to pass these parameters to the remote process Go in space*/
    mmap_addr  =  GetRemoteFuncAddr ( pid ,  libc_path ,  ( void  * ) mmap );
    LOGD ( "mmap RemoteFuncAddr:0x%lx" ,  ( long ) mmap_addr );

    //Parameter
    parameters [ 0 ]  =  0 ; //Setting NULL means let the system choose the memory location for
allocation
    parameters [ 1 ]  =  0x1000 ; //Allocate memory space size to 1 memory page
    parameters [ 2 ]  =  PROT_READ  |  PROT_WRITE  |  PROT_EXEC ; //The permissions of the allocated
memory area are readable, writable, and executable
    parameters [ 3 ]  =   MAP_ANONYMOUS  |  MAP_PRIVATE ; //Anonymous mapping, which means that it is not
supported by files, the following two parameters can be 0
    parameters [ 4 ]  =  0 ;//File identifier, 0 here means no file content mapping
    parameters [ 5 ]  =  0 ; //File mapping offset

    //调用mmap函数
    if (ptrace_call(pid, (long)mmap_addr, parameters, 6, &CurrentRegs) == -1)
    {
        LOGD("Call Remote mmap Func Failed");
        ptrace_detach(pid);
        return iRet;
    }

    //Get the address of the allocated memory area
    RemoteMapMemoryAddr  =  ( void  * ) ptrace_getret ( & CurrentRegs );
    LOGD ( "Remote Process Map Memory Addr:0x%lx" ,  ( long ) RemoteMapMemoryAddr );

    /* 4. Let the remote process execute dlopen to load the so library into the memory.
     * Here you need to first pass the so library path in the dlopen parameter to the memory space of the
remote process, so that it can get the corresponding constant value from its own memory space when it
calls dlopen*/

    //Write the so library path in the newly opened memory space of the remote process
    //Write the so library path in the newly opened memory space of the remote process
    if  ( ptrace_writedata ( pid ,  RemoteMapMemoryAddr ,  LibPath ,  strlen ( LibPath )  +  1 )  ==  - 1
)
    {
        LOGD ( "Write LibPath:%s to RemoteProcess error " ,  LibPath );
        ptrace_detach ( pid );
        return  iRet ;
    }

    //参数
    parameters[0] = (long)RemoteMapMemoryAddr;
    parameters[1] = RTLD_NOW| RTLD_GLOBAL;

    dlopen_addr = GetRemoteFuncAddr(pid, linker_path, (void *)dlopen);
    LOGD("dlopen RemoteFuncAddr:0x%lx", (long)dlopen_addr);
    dlerror_addr = GetRemoteFuncAddr(pid, linker_path, (void *)dlerror);
    LOGD("dlerror RemoteFuncAddr:0x%lx", (long)dlerror_addr);
    dlclose_addr = GetRemoteFuncAddr(pid, linker_path, (void *)dlclose);
    LOGD("dlclose RemoteFuncAddr:0x%lx", (long)dlclose_addr);
    if (ptrace_call(pid, (long)dlopen_addr, parameters, 2, &CurrentRegs) == -1)
    {
        LOGD("Call Remote dlopen Func Failed");
        ptrace_detach(pid);
        return iRet;
    }

    //Get the address of the module loaded into the remote process memory
    RemoteModuleAddr  =  ( void  * ) ptrace_getret ( & CurrentRegs );
    LOGD ( "Remote Process load module Addr:0x%lx" ,  ( long ) RemoteModuleAddr );

    // dlopen 错误
    if ((long)RemoteModuleAddr == 0x0)
    {
        LOGD("dlopen error");
        if (ptrace_call(pid, (long)dlerror_addr, parameters, 0, &CurrentRegs) == -1)
        {
            LOGD("Call Remote dlerror Func Failed");
            ptrace_detach(pid);
            return iRet;
        }
        char *Error = (void *)ptrace_getret(&CurrentRegs);
        char LocalErrorInfo[1024] = {0};
        ptrace_readdata(pid, Error, LocalErrorInfo, 1024);
        LOGD("dlopen error:%s", LocalErrorInfo);
        ptrace_detach(pid);
        return iRet;
    }

    /* 5. The remote process calls the function loaded into the module.
     * First pass the parameters into the remote process space, then use the dlsym function to search for
the function location, and finally make a call */
    if  ( ptrace_writedata ( pid ,  RemoteMapMemoryAddr  +  strlen ( LibPath )  +  2 ,  FunctionName ,
strlen ( FunctionName )  +  1 )  ==  - . 1 )
    {
        LOGD ( "the Write FunctionName:% S to RemoteProcess error" ,  FunctionName );
        ptrace_detach ( PID );
        return  iRet ;
    }

    //设置dlsym参数
    parameters[0] = (long)RemoteModuleAddr;
    parameters[1] = (long)(RemoteMapMemoryAddr + strlen(LibPath) + 2);
    LOGD("Func Name: %x\n", parameters[1]);

    // call the function and get dlsym function returns the address
    dlsym_addr  =  GetRemoteFuncAddr ( pid ,  linker_path ,  ( void  * ) dlsym );
    LOGD ( "dlsym RemoteFuncAddr: 0x% LX" ,  ( Long ) dlsym_addr );
    IF  ( ptrace_call ( pid ,  ( long ) dlsym_addr ,  parameters ,  2 ,  & CurrentRegs )  ==  - 1 )
    {
        LOGD ("Call Remote dlsym Func Failed");
        ptrace_detach(pid);
        return iRet;
    }
    RemoteModuleFuncAddr = (void *)ptrace_getret(&CurrentRegs);
    LOGD("Remote Process ModuleFunc Addr:0x%lx", (long)RemoteModuleFuncAddr);

    /* 6. Call the function loaded into the module in the remote process. For the sake of simplicity, the
    /* 6. Call the function loaded into the module in the remote process. For the sake of simplicity, the
parameter is not selected here, so the step of writing the parameter to the remote access space is
omitted.*/
    if  ( ptrace_call ( pid ,  ( long ) RemoteModuleFuncAddr ,  FuncParameter ,  NumParameter ,  &
CurrentRegs )  ==  - 1 )
    {
        LOGD ( "Call Remote injected Func Failed" );
        ptrace_detach ( pid );
        return  iRet ;
    }

    /* 7. Resume the execution of the remote process*/
    if  ( ptrace_setregs ( pid ,  & OriginalRegs )  ==  - 1 )
    {
        LOGD ( "Recover reges failed" );
        ptrace_detach ( pid );
        return  iRet ;
    }
    LOGD ( "Recover Regs Success" );
    ptrace_getregs ( pid ,  & CurrentRegs );
    if  ( memcmp ( & OriginalRegs ,  & CurrentRegs , sizeof(CurrentRegs)) != 0)
    {
        LOGD("Set Regs Error");
    }

    if (ptrace_detach(pid) == -1)
    {
        LOGD("ptrace detach failed");
        return iRet;
    }

    return 0;
}
```

Entry file of the injection tool : InjectModule.c

```
/************************************************************
  FileName: InjectModule.c
  Description:      ptrace注入
*********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/user.h>
#include <asm/ptrace.h>
#include <sys/ptrace.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <dlfcn.h>
#include <dirent.h>
#include <unistd.h>
#include <string.h>
#include <elf.h>
#include <ptraceInject.h>
#include <PrintLog.h>

/*************************************************
  Description: The PID of the process located by the process name
  Input: process_name is the name of the process to be located
  Output: None
  Return: Return the PID of the located process, if it is -1, it means the positioning failed.
  Others: None
******* ****************************************/
pid_t  FindPidByProcessName ( const  char  * process_name )
{
    int  ProcessDirID  =  0 ;
    pid_t  PID  =  - . 1 ;
    the FILE  * FP  =  NULL ;
    char  filename [ the MAX_PATH ] = {0};
    char cmdline[MAX_PATH] = {0};

    struct dirent * entry = NULL;

    if ( process_name == NULL )
        return -1;

    DIR* dir = opendir( "/proc" );
    if ( dir == NULL )
        return -1;

    while( (entry = readdir(dir)) != NULL )
    {
        ProcessDirID = atoi( entry->d_name );
        if ( ProcessDirID != 0 )
        {
            snprintf(filename, MAX_PATH, "/proc/%d/cmdline", ProcessDirID);
            fp = fopen( filename, "r" );
            if ( fp )
            {
                fgets(cmdline, sizeof(cmdline), fp);
                fclose(fp);

                if (strncmp(process_name, cmdline, strlen(process_name)) == 0)
                {
                    pid = ProcessDirID;
                    break;
                }
            }
        }
    }

    closedir(dir);
    return pid;
}

int  main ( int  argc ,  char  * argv [])  {
    char  InjectModuleName [ MAX_PATH ]  =  "/data/libInjectModule.so" ;     // The full path of the
injected module
    char  RemoteCallFunc [ MAX_PATH ]  =  "Inject_entry" ;               // Called after injecting the
module Module function name
```

```
    char  InjectProcessName [ MAX_PATH ]  =  "com.testjni" ;                    // Inject process name

    // 当前设备环境判断
    #if defined(__i386__)
    LOGD("Current Environment x86");
    return -1;
    #elif defined(__arm__)
    LOGD("Current Environment ARM");
    #else
    LOGD("other Environment");
    return -1;
    #endif

    pid_t pid = FindPidByProcessName(InjectProcessName);
    if (pid == -1)
    {
        printf("Get Pid Failed");
        return -1;
    }

    printf("begin inject process, RemoteProcess pid:%d, InjectModuleName:%s, RemoteCallFunc:%s\n", pid,
InjectModuleName, RemoteCallFunc);
    int iRet = inject_remote_process(pid,  InjectModuleName, RemoteCallFunc,  NULL, 0);
    //int iRet = inject_remote_process_shellcode(pid,  InjectModuleName, RemoteCallFunc,  NULL, 0);

    if (iRet == 0)
    {
        printf("Inject Success\n");
    }
    else
    {
        printf("Inject Failed\n");
    }
    printf("end inject,%d\n", pid);
    return 0;
}
```

Compilation process

Compile the above injected code into an executable file

Preparing Files

Injection tool

- jni folder

  The compiled file must be in the jni folder

- jni/Android.mk

  Compile the configuration file.

  - LOCAL_PATH specifies the directory where the source file is located

  - CLEAR *VARS is used to clear many LOCAL* * variables (but will not clear LOCAL_PATH), because if you compile multiple so you need to clear the configuration information of the previous so file

  - LOCAL_MODULE: The name of the compiled module, with .so appended by default

  - LOCAL_SRC_FILES: compiled source files

  - LOCAL_LDLIBS: The source file uses some system libraries, which are linked to specific locations through this variable tag. For example `-L$(SYSROOT)/usr/lib` , the directory location of the specified standard library is located in sysroot/usr/include under the NDK installation root directory.

    And `-llog` it marks the use of liblog.so log library

    - `include $(BUILD_EXECUTABLE)` Then specify to compile into an executable file

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

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

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

include $(BUILD_EXECUTABLE)
```

- jni/Application.mk

  Specify to generate files suitable for 32-bit ARMv7 instruction set

```
APP_ABI := armeabi-v7a
```

- Source files of the injection tool: InjectModule.c, ptraceInject.c, ptraceInject.h, PrintLog.h
    - InjectModule.c: injection tool entry file
    - ptraceInject.c: the function code of the injection tool
    - ptraceInject.h: The header file that declares the injection function
    - PrintLog.h: The header file that declares the log function

Demo module injected

- jni folder
- jni/Android.mk

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

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

LOCAL_ARM_MODE := arm

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

include $(BUILD_SHARED_LIBRARY)
```

- jni/Application.mk

```
APP_ABI := armeabi-v7a
```

- The source file InjectModule.c of the injected module, and the log print header file PrintLog.h, this header file is the same as above

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

int Inject_entry()
{
    LOGD("Inject_entry Func is called\n");
    return 0;
}
```

Compile

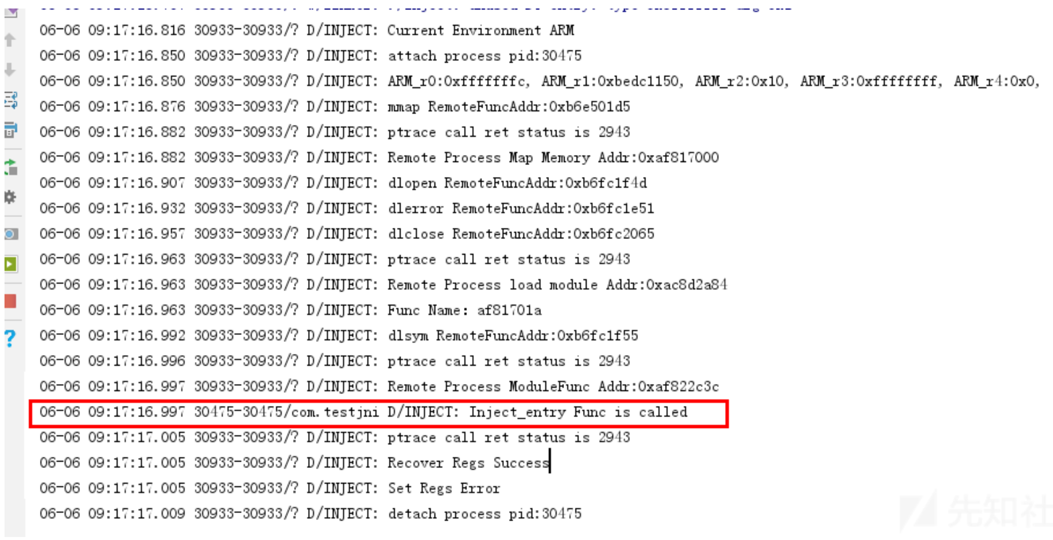
Execute `ndk-build` commands in the two jni directories respectively , you need to configure environment variables

Generated location: In the libs/armeabi-v7a folder in the same directory as jni

Injection process

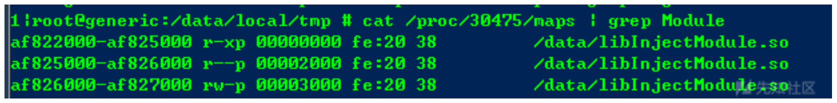
Put the compiled `libInjectModule.so` file into the /data directory, then chmod 777 provides execution permissions to the injection tool, and run it to see in the log that the injected application executes the function in the libInjectModule.so module we injected.

Of course, you can also check the memory layout of the process, you can see that we



```
06-06 09:17:16.816 30933-30933/? D/INJECT: Current Environment ARM
06-06 09:17:16.850 30933-30933/? D/INJECT: attach process pid:30475
06-06 09:17:16.850 30933-30933/? D/INJECT: ARM_r0:0xfffffffc, ARM_r1:0xbedc1150, ARM_r2:0x10, ARM_r3:0xffffffff, ARM_r4:0x0,
06-06 09:17:16.876 30933-30933/? D/INJECT: mmap RemoteFuncAddr:0xb6e501d5
06-06 09:17:16.882 30933-30933/? D/INJECT: ptrace call ret status is 2943
06-06 09:17:16.882 30933-30933/? D/INJECT: Remote Process Map Memory Addr:0xaf817000
06-06 09:17:16.907 30933-30933/? D/INJECT: dlopen RemoteFuncAddr:0xb6fc1f4d
06-06 09:17:16.932 30933-30933/? D/INJECT: dlerror RemoteFuncAddr:0xb6fc1e51
06-06 09:17:16.957 30933-30933/? D/INJECT: dlclose RemoteFuncAddr:0xb6fc2065
06-06 09:17:16.963 30933-30933/? D/INJECT: ptrace call ret status is 2943
06-06 09:17:16.963 30933-30933/? D/INJECT: Remote Process load module Addr:0xac8d2a84
06-06 09:17:16.963 30933-30933/? D/INJECT: Func Name: af81701a
06-06 09:17:16.992 30933-30933/? D/INJECT: dlsym RemoteFuncAddr:0xb6fc1f55
06-06 09:17:16.996 30933-30933/? D/INJECT: ptrace call ret status is 2943
06-06 09:17:16.997 30933-30933/? D/INJECT: Remote Process ModuleFunc Addr:0xaf822c3c
06-06 09:17:16.997 30475-30475/com.testjni D/INJECT: Inject_entry Func is called
06-06 09:17:17.005 30933-30933/? D/INJECT: ptrace call ret status is 2943
06-06 09:17:17.005 30933-30933/? D/INJECT: Recover Regs Success
06-06 09:17:17.005 30933-30933/? D/INJECT: Set Regs Error
06-06 09:17:17.009 30933-30933/? D/INJECT: detach process pid:30475
```

(https://xzfile.aliyuncs.com/media/upload/picture/20190606213139-69e89dd8-885f-1.png)



```
1|root@generic:/data/local/tmp # cat /proc/30475/maps | grep Module
af822000-af825000 r-xp 00000000 fe:20 38         /data/libInjectModule.so
af825000-af826000 r--p 00002000 fe:20 38         /data/libInjectModule.so
af826000-af827000 rw-p 00003000 fe:20 38         /data/libInjectModule.so
```

(https://xzfile.aliyuncs.com/media/upload/picture/20190606213126-621bfbf4-885f-1.png)

summary

Inject our own written so library through dlopen and dlsym functions, remember a few points

[1] As long as the remote process calls dlopen to open the so file we have written, the so library is injected, and then the function address in the library can be retrieved by passing the function name and the handle opened by dlopen as parameters to dlsym.

[2] The path and function name of the so file used above need to be passed to the memory space of the remote process. The remote process cannot access the string in our injection tool across processes.

[3] The call of the remote process is mainly realized by modifying the register through ptrace, pc determines which instruction the function calls, r0-r3 and stack determine the parameters

refer to

[Book] Game Security-Introduction to Mobile Game Security Technology

2 replies

281891****@qq.co (/u/36949) **2020-10-04 00:26:40**

Why do I get all 255 from remote memory read

0 Reply Ta

281891****@qq.co (/u/36949) **2020-10-04 01:43:06**

Hello, I am detected when I read and write mem to the game, how can I prevent the detection?

0 Reply Ta

Log (https://account.aliyun.com/login/login.htm?oauth_callback=https%3A%2F%2Fxz.aliyun.com%2Ft%2F5361&from_type=xianzhi) in to

follow up

Prophet Community

Log in now (https://account.aliyun.co

漏洞情报奖励 X 计划

(/t/7739)

Community blackboard (/notice)

Annual contrib... Monthly contri...

WHOAMIBunny (/u/31...	3
17 year old one (/u/3...	2
J01n (/u/49754)	2
PURE (/u/12865)	2
saulGoodman (/u/181...	2

content