ANDROID Debugging and testing technology compilation

1. IDA Debug port detection

principle:

The debugger will occupy some fixed port numbers when debugging remotely.

practice:

Read/ proc/net/tcp , Find IDA Used for remote debugging 23946 Port, if found, the process is being IDA debugging.

(You can also run netstat apn Search in results 23946 port)

```
void CheckPort23946ByTcp ()
{
	FILE* pfile= NULL ;
	char buf[ 0x1000 ]={ 0 };
	// Excuting an order
	char * strCatTcp= "cat /proc/net/tcp |grep :5D8A" ;
	//char* strNetstat="netstat |grep :23946";
	pfile=popen(strCatTcp, "r" );
	if ( NULL ==pfile)
	{
		LOGA( "CheckPort23946ByTcp popen Open command failed!\ n" );
		return ;
	}
	// Get results
	while (fgets(buf, sizeof (buf),pfile))
	{
		// Execution to here, judged as debugging state
		LOGA( " carried out cat /proc/net/tcp |grep :5D8A the result of:\ n" ); LOGB( "%s" ,buf);

	} //while
	pclose(pfile);
}
```

2. Debugger process name detection

principle:

Remote debugging should be run on the phone android_server gdbserver gdb Wait for the process.

practice:

Traverse the process, find the fixed process name, and find that the debugger is running.

```
void SearchObjProcess ()
{
        FILE* pfile= NULL ;
        char buf[ 0x1000 ]={ 0 };
        // Excuting an order
        //pfile=popen("ps | awk'{print $9}'","r"); // Partially not supported awk command
        pfile=popen( "ps" , "r" );
        if ( NULL ==pfile)
        {
                LOGA( "SearchObjProcess popen Open command failed!\ n" );
                return ;
        }
        // Get results
        LOGA( "popen Program:\ n" );
        while (fgets(buf, sizeof (buf),pfile))
        {
                // Printing process
                LOGB( " Traverse process:% s\n" ,buf);
                // Find substring
                char * strA= NULL ,strB= NULL ,strC= NULL ,strD= NULL ;
                strA= strstr (buf, "android_server" );
                strB= strstr (buf, "gdbserver" );
                strC= strstr (buf, "gdb" );
                strD= strstr (buf, "fuwu" );
                if (strA || strB ||strC || strD) {

                        // Execution to here, judged as debugging state
                        LOGB( " Found target process:% s\n" ,buf);
                } //if
        } //while
        pclose(pfile);
}
```

3 Parent process name detection

principle:

Sometimes not used apk Reverse the method of additional debugging, but write one. out Load executable files directly so get on

Debug, so that the parent process name of the program and normal startup apk The parent process name is not the same.

The test found:

(1) Normally started apk Program: The parent process is zygote

(2) Debug started apk Procedure: in AS Used in LLDB Debugging found that the parent process is still zygote

(3) Additional debugging apk Program: The parent process is zygote

(4) vs Executable file loading for remote debugging so: The parent process is named gdbserver

Conclusion: The parent process name is not zygote , It is judged as debugging state.

practice:

Read/ proc/pid/cmdline To see if the content is zygote

```
void CheckParents ()
{
	//////////////////
	// Set up buf
	char strPpidCmdline[ 0x100 ]={ 0 };
	snprintf (strPpidCmdline, sizeof (strPpidCmdline), "/proc/%d/cmdl ine" , getppid());

	// open a file
	int file=open(strPpidCmdline,O_RDONLY);
	if (file< 0 )
	{
		LOGA( "CheckParents open error!\ n" );
		return ;
	}
	// File content read into memory
	memset (strPpidCmdline, 0 , sizeof (strPpidCmdline));
	ssize_t ret=read(file,strPpidCmdline, sizeof (strPpidCmdline));
	if (- 1 ==ret)
	{
		LOGA( "CheckParents read error!\ n" );
		return ;
	}
	// Not found return 0
	char sRet= strstr (strPpidCmdline, "zygote" );
	if ( NULL ==sRet)
	{
		// Execution to here, judged as debugging state
		LOGA( " Parent process cmdline No zygote Substring!\ n" );
		return ;
	}
	int i= 0 ;
	return ;
}
```

4 Own process name detection

principle:

Just like the previous article, I also write one. out load so The scene of shelling,

The normal process name is generally apk of com.xxx Such a format.

Code:

slightly

5 apk Thread detection

principle:

same. out load so The scene of shelling,

normal apk The process generally has more than a dozen threads running (for example, there will be jdwp Thread),

Write your own executable file to load so Generally there is only one thread,

The debugging environment can be tested based on this difference.

```
void CheckTaskCount ()
{
	char buf[ 0x100 ]={ 0 };
	char * str= "/proc/%d/task" ;
	snprintf (buf, sizeof (buf),str,getpid());
	// open Directory:
	DIR* pdir = opendir(buf);
	if (!pdir)
	{
		perror( "CheckTaskCount open() fail.\n" );
		return ;
	}
	// View the number of files in the directory:
	struct dirent* pde= NULL ;
	int Count= 0 ;
	while ((pde = readdir(pdir))) {

		// Character filter
		if ((pde->d_name[ 0 ] <= '9' ) && (pde->d_name[ 0 ] >= '0' )) {

			+ + Count;
			LOGB( "%d Thread name:% s\n" ,Count,pde->d_name);
		}
	}
	LOGB( " The number of threads is:% d" ,Count);
	if ( 1 >=Count)
	{
		// It is judged as the debugging state here.
		LOGA( " Debugging status!\ n" );
	}
	int i= 0 ;
	return ;
}
```

6 apk process fd File detection

principle:

according to/ proc/pid/fd/ The difference in the number of files under the path to determine the process status.

(apk Started process and non apk Started process fd Different quantity)

(apk of debug Startup and normal startup, process fd The quantity is also different)

Code:

slightly

7 Android system comes with debug detection function

// android.os.Debug.isDebuggerConnected();

principle:

analysis android Built-in debugging detection function isDebuggerConnected() in native The realization of,

Try at native use.

practice:

(1) dalvik In mode:

Find process libdvm.so middle dvmDbgIsDebuggerConnected() function,

Call him to know whether the program is debugged.

dlopen(/system/lib/libdvm.so)

dlsym(_Z25dvmDbgIsDebuggerConnectedv)

(2) art In mode:

art Mode, the results are stored in libart.so Global variables gDebuggerActive in,

The symbol name is _ ZN3art3Dbg15gDebuggerActiveE .

But it looks like a new version android Non- ndk Native library, dlopen(libart.so) Will fail.

So unusable dalvik That way.

There is a troublesome way to manually search in the memory libart Module, and then manually find the global variable symbol.

```
// Only wrote dalvik Code, art Don't write
typedef unsigned char wbool;
typedef wbool (*PPP)();
void NativeIsDBGConnected ()
{
        void * Handle= NULL ;
        Handle=dlopen( "/system/lib/libdvm.so" , RTLD_LAZY);
        if ( NULL ==Handle)
        {
                LOGA( "dlopen turn on libdvm.so failure!\ n" );
                return ;
        }
        PPP Fun = (PPP)dlsym(Handle, "_Z25dvmDbgIsDebuggerConnectedv" );
        if ( NULL ==Fun)
        {
                LOGA( "dlsym Obtain_ Z25dvmDbgIsDebuggerConnectedv failure!\ n" );
                return ;
        }
        else
        {
                wbool ret = Fun();
                if ( 1 ==ret)
                {
                        // It is judged as debug mode here
                        LOGA( "dalvikm Mode, debugging state!\ n" );
                        return ;
                }
        }
        return ;
}
```

8 ptrace Detect

principle:

Each process can only be 1 Debugging process ptrace ,once again p I will fail.

practice:

1 initiative ptrace For yourself, judge whether you have been debugged based on the return value.

2 Or multi-process ptrace .

```
// Single thread ptrace
void ptraceCheck ()
{
        // ptrace If it is debugged, the return value is- 1 , If it runs normally, the return value is 0
        int iRet=ptrace(PTRACE_TRACEME, 0 , 0 , 0 );
        if (- 1 == iRet)
        {
                LOGA( "ptrace Failed, the process is being debugged\ n" );
                return ;
        }
        else
        {
                LOGB( "ptrace The return value is:% d\n" ,iRet);
                return ;
        }
}
```

9 function hash Value detection

principle:

so The instruction of the function in the file is fixed, but if a software breakpoint is set, the instruction will change (the breakpoint address is changed)

Written as bkpt Breakpoint instructions), you can calculate the number of instructions in the memory hash Value is checked to check whether the function has been modified or

The breakpoint was set.

Code:

slightly

10 Breakpoint instruction detection

principle:

As mentioned above, if the function is broken by the software, the breakpoint address will be rewritten as bkpt instruction,

You can search in the function body bkpt Instructions to detect software power failure.

```
// IDA 6.8 Breakpoint scan
// parameter 1 : The first address parameter of the function 2 :function size
typedef uint8_t u8;
typedef uint32_t u32;
void checkbkpt (u8* addr,u32 size) {

	// result
	u32 uRet= 0 ;
	// Breakpoint instruction
//		u8 armBkpt[4]={0xf0,0x01,0xf0,0xe7};
//		u8 thumbBkpt[2]={0x10,0xde};
	u8 armBkpt[ 4 ]={ 0 };
	armBkpt[ 0 ]= 0xf0 ;
	armBkpt[ 1 ]= 0x01 ;
	armBkpt[ 2 ]= 0xf0 ;
	armBkpt[ 3 ]= 0xe7 ;
	u8 thumbBkpt[ 2 ]={ 0 };
	thumbBkpt[ 0 ]= 0x10 ;
	thumbBkpt[ 1 ]= 0xde ;
	// Judgment Mode
	int mode=(u32)addr% 2 ;
	if ( 1 ==mode) {
		LOGA( "checkbkpt:(thumb mode) The address is thumb mode\ n" );
		u8* start=(u8*)((u32)addr- 1 );
		u8* end=(u8*)((u32)start+size);
		// Traversal comparison
		while ( 1 )
		{
			if (start >= end) {
				uRet= 0 ;
				LOGA( "checkbkpt:(no find bkpt) No breakpoint was found.\ n" );
				break ;
			}
			if ( 0 == memcmp (start,thumbBkpt, 2 )) {uRet= 1 ;

				LOGA( "checkbkpt:(find it) A breakpoint was found.\ n" );
				break ;
			}
			start=start+ 2 ;
		} //while
	} //if
	else
	{
		LOGA( "checkbkpt:(arm mode) The address is arm mode\ n" );
		u8* start=(u8*)addr;
		u8* end=(u8*)((u32)start+size);
		// Traversal comparison
		while ( 1 )
```

```
                {
                        if (start >= end) {
                                uRet = 0 ;
                                LOGA( "checkbkpt:(no find) No breakpoint was found.\ n" );
                                break ;
                        }
                        if ( 0 == memcmp (start,armBkpt, 4 )){ uRet = 1 ;

                                LOGA( "checkbkpt:(find it) A breakpoint was found.\ n" );
                                break ;
                        }
                        start = start + 4 ;} //while

        } //else
        return ;
}
```

11 System source code modification detection

principle:

Android native The popular anti-debugging solution is to read the process status or stat To detect tracepid , The principle is debugging status

Process tracepid Not for 0 .

For this debugging detection method, a thorough bypass is to modify the system source code and recompile, so that tracepid Always for

0 .

Against this bypass Means, we can create a child process, let the child process take the initiative ptrace Set itself to debug state,

At this time, under normal circumstances, the child process tracepid Should not be 0 . At this point we detect the child process tracepid Is it 0 ,

If 0 That the source code has been modified.

```
bool checkSystem ()
{
        // Build pipeline
        int pipefd[ 2 ];
        if (- 1 == pipe(pipefd)){
                LOGA( "pipe() error.\n" );
                return false ;
        }
        // Create child process
        pid_t pid = fork(); LOGB( "father pid is: %d\n" ,getpid()); LOGB( "child
        pid is: %d\n" ,pid);

        // for failure
        if ( 0 > pid) {
                LOGA( "fork() error.\n" );
                return false ;
        }
        // Subprocess program
        int childTracePid= 0 ;
        if ( 0 == pid)
        {
                int iRet = ptrace(PTRACE_TRACEME, 0 , 0 , 0 );
                if (- 1 == iRet)
                {
                        LOGA( "child ptrace failed.\n" );
                        exit ( 0 );
                }
                LOGA( "%s ptrace succeed.\n" );
                // Obtain tracepid
                char pathbuf[ 0x100 ] = { 0 };
                char readbuf[ 100 ] = { 0 };
                sprintf (pathbuf, "/proc/%d/status" , getpid());
                int fd = openat( NULL , pathbuf, O_RDONLY);
                if (- 1 == fd) {
                        LOGA( "openat failed.\n" );
                }
                read(fd, readbuf, 100 );
                close(fd);
                uint8_t *start = ( uint8_t *) readbuf;
                uint8_t des[ 100 ] = { 0x54 , 0x72 , 0x61 , 0x63 , 0x65 , 0x72 , 0x5 0 , 0x69 , 0x64 , 0x3A , 0x09 };

                int i               = 100 ;
                bool flag= false ;
                while (--i)
                {
                        if ( 0 == memcmp (start,des, 10 ))
                        {
                                start=start+ 11 ;
```

```
                childTracePid=atoi(( char *)start);
                flag= true ;
                break ;
            } else
            {
                start=start+ 1 ;
                flag= false ;
            }
        } //while
        if ( false ==flag) {
            LOGA( "get tracepid failed.\n" );
            return false ;
        }
        // Write data to the pipeline
        close(pipefd[ 0 ]);                                    // Close the pipe read end
        write(pipefd[ 1 ], ( void *)&childTracePid, 4 ); // Write to the pipe writer
data
        close(pipefd[ 1 ]);                                    // Close the pipe to write after writing
end
        LOGA( "child succeed, Finish.\n" );
        exit ( 0 );
    }
    else
    {
        // Parent process program
        LOGA( " Start waiting for the child process.\ n" );
        waitpid(pid, NULL , NULL );                            // Wait for child process
End
        int buf2 = 0 ;
        close(pipefd[ 1 ]);                                    // Close the writer
        read(pipefd[ 0 ], ( void *)&buf2, 4 );                 // Read from the reader
Data to buf
        close(pipefd[ 0 ]);                                    // Close the reader
        LOGB( " The content passed by the child process is:% d\n" , buf2);     // Output content
        // Judging the child process ptarce After tracepid
        if ( 0 == buf2) {
            LOGA( " The source code has been modified.\ n" );
        } else {
            LOGA( " The source code has not been modified.\ n" );
        }
        return true ;
    }
}
void smain ()
{
    bool bRet=checkSystem();
    if ( true ==bRet)
        LOGA( "check succeed.\n" );
    else
```

```
                LOGA( "check failed.\n" );
        LOGB( "main Finish pid:%d\n" ,getpid());
        return ;
    }
```

12 Single step debugging trap

principle:

Process analysis of the debugger from the next breakpoint to the execution of the breakpoint:

1 Save: save the target instruction

2 Replace: Replace the instruction at the target with a breakpoint instruction

3 Hit break point: hit the break point instruction (cause an interrupt or send a signal)

4 Received signal: After the debugger receives the signal, it executes the signal processing function registered by the debugger.

5 Restore: The debugger processing function restores the saved instructions

6 Rollback: Rollback PC register

7 Control return procedure.

Anti-debugging scheme for actively setting breakpoint instructions/registering signal processing functions:

1 Write breakpoint instructions in the function

2 Register the breakpoint signal processing function in the code

3 The program executes to a breakpoint instruction and sends a signal

There are two situations:

(1) Non-debugging state

Enter the function registered by yourself, NOP Instruction replaces breakpoint instruction, rewind PC After normal instructions.

(Execute a breakpoint to signal — Enter the signal processing function —NOP Replace breakpoint — return PC)

(2) Debug status

Enter the breakpoint processing flow of the debugger, he will recover the failure of the instruction at the target, and then roll back PC , Into an endless loop.

```
# !cpp
char dynamic_ccode[] = { 0x1f , 0xb4 , //push {r0-r4}
                                                0x01 , 0xde , //breakpoint
                                                0x1f , 0xbc , //pop {r0-r4}
                                                0xf7 , 0x46 }; //mov pc,lr

char *g_addr = 0 ;

void my_sigtrap ( int sig){

        char change_bkp[] = { 0x00 , 0x46 }; //mov r0,r0
        memcpy (g_addr+ 2 ,change_bkp, 2 );
        __clear_cache(( void *)g_addr,( void *)(g_addr+ 8 )); // need to clea r cache

        LOGI( "chang bpk to nop\n" );

}

void anti4 (){ //SIGTRAP

        int ret,size;
        char *addr,*tmpaddr;

        signal(SIGTRAP,my_sigtrap);

        addr = ( char *) malloc (PAGESIZE* 2 );

        memset (addr, 0 ,PAGESIZE* 2 );
        g_addr = ( char *)((( int ) addr + PAGESIZE- 1 ) & ~(PAGESIZE- 1 ));

        LOGI( "addr: %p ,g_addr: %p\n" ,addr,g_addr);

        ret = mprotect(g_addr,PAGESIZE,PROT_READ|PROT_WRITE|PROT_EXEC);
        if (ret!= 0 )
        {
                LOGI( "mprotect error\n" );
                return ;
        }

        size = 8 ;
        memcpy (g_addr,dynamic_ccode,size);

        __clear_cache(( void *)g_addr,( void *)(g_addr+size)); // need to c lear cache


        __asm__( "push {r0-r4,lr}\n\t"
                        "mov r0,pc\n\t" // at this time pc Point to the next two instructions
                        "add r0,r0,#4\n\t" //+4 Yes lr Address is pop{r0-r5}
```

```
                        "mov lr,r0\n\t"
                        "mov pc,%0\n\t"
                        "pop {r0-r5}\n\t"
                        "mov lr,r5\n\t" // restore lr
        :
        : "r" (g_addr)
        :);

        LOGI( "hi, i'm here\n" );
        free (addr);

}
```

13 use IDA First intercept signal characteristic detection

principle:

IDA The signal will be intercepted first, causing the process to fail to receive the signal, and the signal processing function will not be executed. Key process

In the signal processing function, if it is not executed, it is in the debug state.

```c
# include <stdio.h>
# include <signal.h>
# include <unistd.h>
void myhandler ( int sig)
{
        //signal(5, myhandler);
        printf ( "myhandler.\n" );
        return ;
}
int g_ret = 0 ;
int main ( int argc, char **argv) {

        // Set up SIGTRAP The signal processing function is myhandler()
        g_ret = ( int )signal(SIGTRAP, myhandler);
        if (( int )SIG_ERR == g_ret)
                printf ( "signal ret value is SIG_ERR.\n" );

        // print signal Return value (original processing function address)
        printf ( "signal ret value is %x\n" ,( unsigned char *)g_ret);

        // Actively send to your own process SIGTRAP signal
        raise(SIGTRAP);
        raise(SIGTRAP);
        raise(SIGTRAP);
        kill(getpid(), SIGTRAP);

        printf ( "main.\n" );
        return 0 ;
}
```

14 use IDA Analyze defects and anti-debug

principle:

IDA The recursive descent algorithm is used to disassemble instructions, and the major disadvantage of this algorithm is that it cannot handle indirect code paths.

The jump calculated dynamically is not recognized. and arm Due to the existence of arm with thumb Instruction set, it involves instruction set

Switch, IDA In some cases, it cannot be recognized arm with thumb Instruction, further resulting in the failure to restore pseudo-code.

in IDA During dynamic debugging, this problem still exists. If a breakpoint is written in a place where the instruction recognizes the error, it may cause debugging

The device crashed. (Maybe write breakpoints, don’t know how to write ARM still is THUMB, Crash caused)

```
# if (JUDGE_THUMB)
# define GETPC_KILL_IDAF5_SKATEBOARD                    \
__asm   __volatile(                                     \
"mov          r0,pc                     \n\t"           \
"adds r0,0x9                            \n\t"           \
"push {r0}                              \n\t"           \
"pop          {r0}                      \n\t"           \
"bx           r0                        \n\t"           \
\
".byte 0x00                             \n\t"           \
".byte 0xBF                             \n\t"           \
\
".byte 0x00                             \n\t"           \
".byte 0xBF                             \n\t"           \
\
".byte 0x00                             \n\t"           \
".byte 0xBF                             \n\t"           \
::: "r0"                                                \
);
# else
# define GETPC_KILL_IDAF5_SKATEBOARD                    \
__asm   __volatile(                                     \
"mov          r0,pc                     \n\t"           \
"add          r0,0x10                   \n\t"           \
"push {r0}                              \n\t"           \
"pop          {r0}                      \n\t"           \
"bx           r0                        \n\t"           \
".int 0xE1A00000                        \n\t"           \
".int 0xE1A00000                        \n\t"           \
".int 0xE1A00000                        \n\t"           \
".int 0xE1A00000                        \n\t"           \
::: "r0"                                                \
);
# endif



// Constant label version
# if (JUDGE_THUMB)
# define IDAF5_CONST_1_2                                        \
__asm   __volatile(                                             \
"b              T1                      \n\t"                   \
"T2:                                    \n\t"                   \
"adds           r0,1                    \n\t"                   \
"bx             r0                      \n\t"                   \
"T1:                                    \n\t"                   \
"mov            r0,pc                   \n\t"                   \
"b              T2                      \n\t"                   \
::: "r0"                                                        \
```

```
);
#else
#define IDAF5_CONST_1_2                                    \
__asm   __volatile(                                        \
"b              T1                      \n\t"              \
"T2:                                    \n\t"              \
"bx             r0                      \n\t"              \
"T1:                                    \n\t"              \
"mov            r0,pc                   \n\t"              \
"b              T2                      \n\t"              \
::: "r0"                                                   \
);
#endif
```

15 Five types of code execution time detection

the first sort:

principle:

A piece of code in a To get the time, run for a while, and then b Get the time,

Then pass (b time a Time) Find the time difference, which will be very small under normal circumstances.

If this time difference is relatively large, it means that it is being stepped through.

practice:

Five can get time api :

time() function

time_t Structure

clock() function

clock_t Structure

gettimeofday() function

timeval structure

timezone structure

clock_gettime() function

timespec structure

getrusage() function

rusage structure

```c
# include <sys/time.h>
# include <sys/types.h>
# include <sys/resource.h>
# include <stdio.h>
# include <stdlib.h>
# include <unistd.h>
# include <time.h>

static int _getrusage ();                       //Invalid
static int _clock ();                           //Invalid
static int _time ();
static int _gettimeofday ();
static int _clock_gettime ();

int main ()
{
        _getrusage ();
        _clock ();
        _time ();
        _gettimeofday ();
        _clock_gettime ();

        return 0 ;
}

int _getrusage ()
{
        struct rusage t1;
        /* breakpoint */
        getrusage (RUSAGE_SELF, &t1);

        long used = t1.ru_utime.tv_sec + t1.ru_stime.tv_sec;
        if (used> 2 ) {
                puts ( "debugged" );
        }
        return 0 ;
}


int _clock ()
{
        clock_t t1, t2;

        t1 = clock ();
        /* breakpoint */
        t2 = clock ();
        double used = ( double )(t2-t1) / CLOCKS_PER_SEC;
        if (used> 2 ) {
```

```
                puts ( "debugged" );
        }
        return 0 ;
}


int _time ()
{
        time_t t1, t2;
        time (&t1);
        /* breakpoint */
        time (&t2);
        if (t2-t1> 2 ) {
                puts ( "debugged" );
        }
        return 0 ;
}

int _gettimeofday ()
{
        struct timeval t1, t2;
        struct timezone t;

        gettimeofday (&t1, &t);
        /* breakpoint */
        gettimeofday (&t2, &t);

        if (t2.tv_sec-t1.tv_sec> 2 ) {
                puts ( "debugged" );
        }
        return 0 ;
}

int _clock_gettime ()
{
        struct timespec t1,   t2;

        clock_gettime (CLOCK_REALTIME, &t1);
        /* breakpoint */
        clock_gettime (CLOCK_REALTIME, &t2);

        if (t2.tv_sec-t1.tv_sec> 2 ) {
                puts ( "debugged" );
        }
        return 0 ;
}
```

16 Three types of process information structure detection

principle:

Some process files store process information, which can be read to know whether it is in a debugging state.

practice:

The first:

/proc/pid/status

/proc/pid/task/pid/status

TracerPid non- 0

statue Write in the field t (tracing stop)

The second type:

/proc/pid/stat

/proc/pid/task/pid/stat

The second field is t (T)

The third type:

/proc/pid/wchan

/proc/pid/task/pid/wchan

ptrace_stop

Code:

slightly.

17 Inotify Event monitoring dump

principle:

Usually the shell will complete the alignment before the program runs text Decryption, so the shelling can be passed dd versus gdb_gcore Come dump

/proc/pid/mem or/ proc/pid/pagemap , Get the decrypted code content.

able to pass Inotify series api To monitor mem or pagemap Open or access events,

Stop the process as soon as the time occurs dump .

```
void thread_watchDumpPagemap ()
{
	LOGA( "------------------watchDump:Pagemap-----------------\n" );

	char dirName[NAME_MAX]={ 0 };
	snprintf (dirName,NAME_MAX, "/proc/%d/pagemap" ,getpid());
	int fd = inotify_init();
	if (fd < 0 )
	{
		LOGA( "inotify_init err.\n" );
		return ;
	}
	int wd = inotify_add_watch(fd,dirName,IN_ALL_EVENTS);
	if (wd < 0 )
	{
		LOGA( "inotify_add_watch err.\n" );
		close(fd);
		return ;
	}
	const int		buflen= sizeof ( struct inotify_event) * 0x100 ; buf[buflen]={ 0 };
	char
	fd_set		readfds;
	while ( 1 )
	{
		FD_ZERO(&readfds);
		FD_SET(fd, &readfds);
		int iRet = select(fd+ 1 ,&readfds, 0 , 0 , 0 ); // Block here
		LOGB( "iRet The return value:% d\n" ,iRet);
		if (- 1 ==iRet)
			break ;
		if (iRet)
		{
			memset (buf, 0 ,buflen);
			int len = read(fd,buf,buflen);
			int i= 0 ;
			while (i <len)
			{
				struct inotify_event *event = ( struct inotify_even
t*)&buf[i];
				LOGB( "1 event mask The value is:% d\n" ,event->mask);
				if ((event->mask==IN_OPEN))
				{
					// Judged here true, Execution crashes.
					LOGB( "2 Someone opened pagemap, % d Times.\ n\n" ,i);
					//__asm	__volatile(".int 0x8c89fa98");
				}
```

```
                                i += sizeof ( struct inotify_event) + event->len;
                        }
                        LOGA( "-----3 Exit the small loop-----\ n" );
                }
        }
        inotify_rm_watch(fd,wd);
        close(fd);
        LOGA( "-----4 Exit the big loop, turn off monitoring -----\ n" );
        return ;
}
void smain ()
{
        // monitor/ proc/pid/mem
        pthread_t ptMem,t,ptPageMap;
        int iRet= 0 ;

        // monitor/ proc/pid/pagemap
        iRet=pthread_create(&ptPageMap, NULL ,(PPP)thread_watchDumpPagema
p, NULL );
        if ( 0 !=iRet)
        {
                LOGA( "Create,thread_watchDumpPagemap,error!\n" );
                return ;
        }
        iRet=pthread_detach(ptPageMap);
        if ( 0 !=iRet)
        {
                LOGA( "pthread_detach,thread_watchDumpPagemap,error!\n" );
                return ;
        }
        LOGA( "------------------smain-------------------\n" );
        LOGB( "pid:%d\n" ,getpid());
        return ;
}
```

by fightclub

Reference:

1 Antidebugging Skills in APK——wooyun 2 Android Escape

technology compilation - 360