

今天我们分析的是一个自写DEMO，快来看看吧。

链接: [https://pan.baidu.com/s/1OhnNhf68Gv7v3JoEhGjs\\_Q](https://pan.baidu.com/s/1OhnNhf68Gv7v3JoEhGjs_Q)

提取码: t0st

```
com.roysue.getrusage.MainActivity X com.roysue.getrusage.R X com.roysue.getrusage.BuildConfig

package com.roysue.getrusage;

import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import androidx.appcompat.app.AppCompatActivity;

12 public class MainActivity extends AppCompatActivity {
    public native int stringFromJNI();

    static {
14         System.loadLibrary("native-lib");
    }

    /* access modifiers changed from: protected */
    @Override // androidx.activity.ComponentActivity, androidx.core.app.ComponentActivity, androidx.fragment.app.Fragment
18 public void onCreate(Bundle savedInstanceState) {
19     super.onCreate(savedInstanceState);
20     setContentView(R.layout.activity_main);
23     final TextView tv = (TextView) findViewById(R.id.sample_text);
25     ((Button) findViewById(R.id.button)).setOnClickListener(new View.OnClickListener() {
        /* class com.roysue.getrusage.MainActivity.AnonymousClass1 */

27         public void onClick(View v) {
28             tv.setText(Integer.toString(MainActivity.this.stringFromJNI()));
        }
    });
}
}
```

这是目标函数，返回值是int，没有入参。因为是demo嘛，所以是静态注册。

```
Instruction Data Unexplored External symbol Lumina function
IDA View-A Pseudocode-A Hex View-1 Structures

1 time_t Java_com_roysue_getrusage_MainActivity_stringFromJNI()
2 {
3     struct rusage usage; // [sp+14h] [bp-54h] BYREF
4
5     getrusage(0, &usage);
6     return usage.ru_utime.tv_sec;
7 }
```

这逻辑也太简单了。getrusage 是libc里的库函数，它用于查看进程的资源消耗情况。

每个进程都会消耗诸如内存和 CPU 时间之类的系统资源。本章将介绍与资源相关的系统调用，首先会介绍 getrusage()系统调用，该函数允许一个进程监控自己及其子进程已经用掉的资源。

看一下函数定义吧

```
int getrusage(int who, struct rusage *usage);
```

它有两个入参，先说参数1，参数1有四个可选的值

```
#define RUSAGE_SELF 0
#define RUSAGE_CHILDREN (- 1)
#define RUSAGE_BOTH (- 2)
#define RUSAGE_THREAD 1
```

0 代表查看当前进程，-1为所有子进程，-2为当前进程+所有子进程，1是线程。

参数2是一个指向 rusage 结构的指针，资源消耗情况返回到该指针指向的结构体

至于函数的返回值，执行成功返回0，发生错误返回 -1。

所以问题就在于rusage这个结构体长啥样，我们就知道这个函数返回什么东西了。

```
struct rusage {
    struct timeval ru_utime; /* user CPU time used */
    struct timeval ru_stime; /* system CPU time used */
    long ru_maxrss; /* maximum resident set size */
    long ru_ixrss; /* integral shared memory size */
    long ru_idrss; /* integral unshared data size */
    long ru_isrss; /* integral unshared stack size */
    long ru_minflt; /* page reclaims (soft page faults) */
    long ru_majflt; /* page faults (hard page faults) */
    long ru_nswap; /* swaps */
    long ru_inblock; /* block input operations */
    long ru_oublock; /* block output operations */
    long ru_msgsnd; /* IPC messages sent */
    long ru_msrgrcv; /* IPC messages received */
    long ru_nsignals; /* signals received */
    long ru_nvcsw; /* voluntary context switches */
    long ru_nivcsw; /* involuntary context switches */
};
```

看着挺复杂的，我们从上往下看

首先是ru\_utime：返回进程在用户模式下的执行时间，以timeval结构的形式返回,timeval一般长这样，

```
struct timeval {
    long tv_sec;
    long tv_usec;
};
```

tv\_sec 是秒，tv\_usec 是微妙。相当于 15块六角五分，前者是15块，后者是六角五分呗。

看一下上图我们的样本，其实就是返回了“秒”。从函数作用上来看，大致可以理解成APP打开了多久。

再是ru\_stime：返回进程在内核模式下的执行时间，以timeval结构的形式返回

和第一个的区别就是用户模式和内核模式，App运行主要是在用户模式下。再略看一下其他的信息

ru\_maxrss: 返回最大驻留集的大小，单位为kb。不太懂是个啥。

ru\_ixrss、ru\_idrss、ru\_isrss: 目前不支持

ru\_minflt: 缺页中断的次数，且处理这些中断不需要进行I/O

ru\_majflt: 缺页中断的次数，且处理这些中断需要进行I/O

ru\_nswap: 目前不支持

ru\_inblock: 文件系统需要进行输入操作的次数。

ru\_oublock: 文件系统需要进行输出操作的次数。

ru\_msgsnd、ru\_msgrcv、ru\_nsignals: 目前不支持

ru\_nvcsw: 因进程自愿放弃处理器时间片而导致的上下文切换的次数（通常是为了等待请求的资源）。

ru\_nivcsw: 因进程时间片使用完毕或被高优先级进程抢断导致的上下文切换的次数。

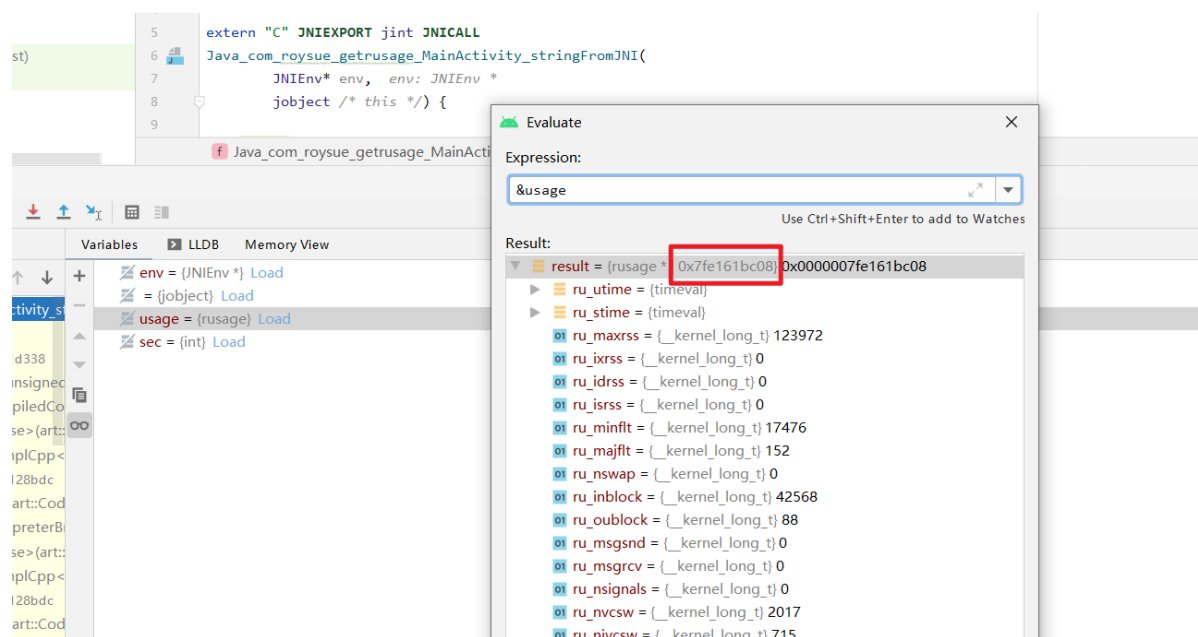
可以发现，确实是查看底层资源的消耗情况，挺复杂的，几乎看不懂是啥。有的人对结构体不熟悉，我们看一下

rusage 这个结果的内存（这个Android Studio 项目就是我们上面的DEMO）



```
1 #include <jni.h>
2 #include <string>
3 #include <sys/resource.h>
4
5 extern "C" JNIEXPORT jint JNICALL
6 Java_com_roysue_getrusage_MainActivity_stringFromJNI(
7     JNIEnv* env,
8     jobject /* this */) {
9
10     struct rusage usage;
11     getrusage(RUSAGE_SELF, &usage);
12     int sec = usage.ru_utime.tv_sec;
13     return sec;
14 }
```

我下了个断点，在此处，usage结构体里是有值的，通过Android Studio 的 LLDB 调试器查看内存，首先获取一下地址



LLDB Variables:

- env = (JNIEnv \*) Load
- usage = (rusage) Load
- sec = (int) Load

Evaluate Expression: &usage

Result:

```
result = {rusage * 0x7fe161bc08} 0x0000007fe161bc08
  ru_utime = (timeval)
  ru_stime = (timeval)
  ru_maxrss = (_kernel_long_t) 123972
  ru_ixrss = (_kernel_long_t) 0
  ru_idrss = (_kernel_long_t) 0
  ru_isrss = (_kernel_long_t) 0
  ru_minflt = (_kernel_long_t) 17476
  ru_majflt = (_kernel_long_t) 152
  ru_nswap = (_kernel_long_t) 0
  ru_inblock = (_kernel_long_t) 42568
  ru_oublock = (_kernel_long_t) 88
  ru_msgsnd = (_kernel_long_t) 0
  ru_msgrcv = (_kernel_long_t) 0
  ru_nsignals = (_kernel_long_t) 0
  ru_nvcsw = (_kernel_long_t) 2017
  ru_nivcsw = (_kernel_long_t) 715
```



大家明白这个DEMO做什么了吗？它调用了资源消耗函数getrusage，查看样本在用户模式下运行的时间，并返回“秒”。

在Unidbg中执行它会遇到什么问题呢？

直接上代码

```
package com.mydemo;

import com.github.unidbg.AndroidEmulator;
import com.github.unidbg.Module;
import com.github.unidbg.linux.android.AndroidEmulatorBuilder;
import com.github.unidbg.linux.android.AndroidResolver;
import com.github.unidbg.linux.android.SystemPropertyHook;
import com.github.unidbg.linux.android.SystemPropertyProvider;
import com.github.unidbg.linux.android.dvm.AbstractJni;
import com.github.unidbg.linux.android.dvm.DalvikModule;
import com.github.unidbg.linux.android.dvm.VM;
import com.github.unidbg.linux.android.dvm.array.ByteArray;
import com.github.unidbg.memory.Memory;

import java.io.File;
import java.util.ArrayList;
import java.util.List;

public class day0813 extends AbstractJni {
    private final AndroidEmulator emulator;
    private final VM vm;
    private final Module module;

    day0813() {
        emulator = AndroidEmulatorBuilder.for32Bit().build();

        final Memory memory = emulator.getMemory(); // 模拟器的内存操作接口
        memory.setLibraryResolver(new AndroidResolver(23)); // 设置系统类库解析

        vm = emulator.createDalvikVM(); // 创建Android虚拟机

        emulator.getSyscallHandler().setVerbose(true);
        vm.setVerbose(true); // 设置是否打印Jni调用细节
        DalvikModule dm = vm.loadLibrary(new File("unidbg-android/src/test/resources/mydemo/libnative-lib.so"), true);
        module = dm.getModule(); //
        vm.setJni(this);
    }

    public int call(){
        List<Object> objectList = new ArrayList<>();
        objectList.add(vm.getJNIEnv());
        objectList.add(0);
        Number number = module.callFunction(emulator, 0x7AD,
        objectList.toArray())[0];
        return number.intValue();
    }

    public static void main(String[] args) {
        day0813 demo = new day0813();
    }
}
```

```
        System.out.println(demo.call());
    }

}
```

运行代码



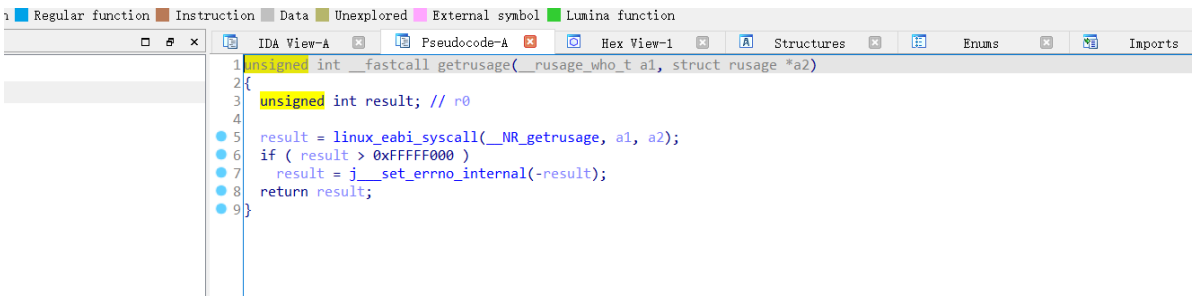
首先，我要强调的是，ARM32SyscallHandler中的日志一定要重视，它代表了系统调用方面所发生的调用或者错误。

NR=77即系统调用号下77，在网站中查一下是哪个系统调用[Chromium OS Docs - Linux System Call Table \(googlesource.com\)](https://chromium.googlesource.com/chromiumos/docs/+master/constants/syscalls.md#arm-32_bit_EABI)

				*filename		
62	ustat	man/ cs/	0x3e	unsigned dev	struct ustat *ubuf	-
63	dup2	man/ cs/	0x3f	unsigned int oldfd	unsigned int newfd	-
64	getppid	man/ cs/	0x40	-	-	-
65	getpgid	man/ cs/	0x41	-	-	-
66	setsid	man/ cs/	0x42	-	-	-
67	sigaction	man/ cs/	0x43	int	const struct old_sigaction *	struct old_sigaction *
68	not implemented		0x44			
69	not implemented		0x45			
70	setreuid	man/ cs/	0x46	uid_t ruid	uid_t euid	-
71	setregid	man/ cs/	0x47	gid_t rgid	gid_t egid	-
72	sigsuspend	man/ cs/	0x48	int unused1	int unused2	old_sigset_t mask
73	sigpending	man/ cs/	0x49	old_sigset_t *uset	-	-
74	sethostname	man/ cs/	0x4a	char *name	int len	-
75	setrlimit	man/ cs/	0x4b	unsigned int resource	struct rlimit *rlim	-
76	not implemented		0x4c			
77	getrusage	man/ cs/	0x4d	int who	struct rusage *ru	-
78	gettimeofday	man/ cs/	0x4e	struct timeval *tv	struct timezone *tz	-
79	settimeofday	man/ cs/	0x4f	struct timeval *tv	struct timezone *tz	-
80	getgroups	man/ cs/	0x50	int gidsetsize	gid_t *grouplist	-
81	setgroups	man/ cs/	0x51	int gidsetsize	gid_t *grouplist	-
82	not implemented		0x52			
83	symlink	man/ cs/	0x53	const char *old	const char *new	-

好家伙，竟然是getrusage，首先我们有个困惑，DEMO里我们没用系统调用啊，分析SO发现getrusage是导入函数，在libc里，这儿怎么又成了系统调用了呢？

事实上，一部分libc里的函数，只是对系统调用的简单封装，比如getrusage函数，下图是libc中的F5代码，就是简单封装罢了。



因此我们使用getrusage相当于间接用了系统调用。

那么问题2，为什么Unidbg会报错？答案也很简单，Unidbg尚未实现这个系统调用

```
case 77:
    return;
case 63:
    backend.reg_write(ArmConst.UC_ARM_REG_R0, dup2(backend, emulator));
    return;
case 64:
    backend.reg_write(ArmConst.UC_ARM_REG_R0, getppid(emulator));
    return;
case 67:
    backend.reg_write(ArmConst.UC_ARM_REG_R0, sigaction(backend, emulator));
    return;
case 74:
    backend.reg_write(ArmConst.UC_ARM_REG_R0, gettimeofday(emulator));
    return;
case 85:
    backend.reg_write(ArmConst.UC_ARM_REG_R0, readlink(emulator));
    return;
case 88:
    backend.reg_write(ArmConst.UC_ARM_REG_R0, reboot(backend, emulator));
    return;
case 91:
    backend.reg_write(ArmConst.UC_ARM_REG_R0, munmap(backend, emulator));
    return;
```

为什么Unidbg没实现这个系统调用呢

- Unidbg作者精力有限，只实现了最常用的系统调用
- 系统调用好几百个，一个一个来嘛
- getrusage 很难有一个完美的实现

getrusage 返回的资源消耗信息，某种程度而言，是不能乱填的，其必然有内在的规律，或许是可以做风控的，所以没办法在Unidbg中进行标准的实现，我是这么认为的emmm。

那我们先进行一个粗糙的实现呗，来看一下吧！

```
case 64:
    backend.reg_write(ArmConst.UC_ARM_REG_R0, getppid(emulator));
    return;
case 67:
    backend.reg_write(ArmConst.UC_ARM_REG_R0, sigaction(backend, emulator));
    return;
case 77:
    backend.reg_write(ArmConst.UC_ARM_REG_R0, getrusage(backend, emulator));
    return;
case 78:
    backend.reg_write(ArmConst.UC_ARM_REG_R0, gettimeofday(emulator));
    return;
case 85:
```

```
private static final int RUSAGE_SELF = 0;
private static final int RUSAGE_CHILDREN = -1;
private static final int RUSAGE_BOTH = -2;
private static final int RUSAGE_THREAD = 1;

private int getrusage(Backend backend, Emulator<?> emulator){
    int who = backend.reg_read(ArmConst.UC_ARM_REG_R0).intValue();
    Pointer usage = UnidbgPointer.register(emulator, ArmConst.UC_ARM_REG_R1);
    usage.setLong(0, 1);
    usage.setLong(8, 0x10000);
    return 0;
};
```

我们这里做了非常粗糙的实现，首先，对于who，即读取模式（主进程、子进程、线程等）情形，没做分门别类的处理，其次，我们也没用给usage结构体的每个内容适合的返回值，只将ru\_utime 的结构体设置成了1秒，0x10000毫秒，其余都没有做处理，即都默认0，这是不合适的。

在真实样本中，建议Frida hook 真实的、样本返回的usage结构体信息，填入此函数中。