对字符串加解密的处理（三）

一、引言

在上一篇中，我们复写了字符串解密算法，根据交叉引用逐个执行，实现对字符串全体的解密，这个思路可以有效避免被动 Dump/Hook 代码覆盖率低导致部分密文未解密的情况，而且不依赖于 Andoid 环境。

但有些时候，我们并不在意方案是否依赖 Android 或其他环境，如果解密算法又比较复杂，分析和复现成本甚至大于字符串解密带来的好处时，这种情况下，可以采用 Frida 或其他框架主动调用解密函数。

二、Frida call 解密函数

这个思路让我们免于对字符串解密做算法上的分析，实践中很受大家的喜爱。我们以第一篇中提到的数美样本做示例。它的调用模式如下

```c
sub_80D8("UWRYubYl2XXvaG3S9r5ezWcxX/VsRigluNW58+nIYq4=", 44LL);
```

函数原型如下

```c
char *__fastcall sub_80D8(char *a1, __int64 a2)
```

大家对 Frida Hook 都很熟悉

JavaScript | 复制代码

```javascript
function hook_sub_80D8(){
    var base_addr = Module.findBaseAddress("libsmsdk.so");

    Interceptor.attach(base_addr.add(0x80d8), {
        onEnter(args) {

        },
        onLeave(retval) {
            console.log(retval.readCString());
        }
    });
}
```

对 Frida Call 应该也不陌生，我们使用它执行函数。

JavaScript | 复制代码

```javascript
// 获取解密函数绝对地址
var base_addr = Module.findBaseAddress("libsmsdk.so");
var decrypt_address = base_addr.add(0x80d8)
// 根据解密函数的定义构造它
var decrypt_Fun = new NativeFunction(decrypt_address, "pointer", ["pointer", '

// 封装对解密函数的调用，并获取返回值
function decrypt(encryptContent, length){
    var encryptAddress = Memory.allocUtf8String(encryptContent);
    var decryptAddress = decrypt_Fun(encryptAddress, length);
    var decryptContent = decryptAddress.readUtf8String();
    console.log("decrypt:"+decryptContent)
}

// 调用测试
decrypt("UWRYubYl2XXvaG3S9r5ezWcxX/VsRigluNW58+nIYq4=", 44)

```

Frida attach 予以测试，发现解密顺利，Frida Call 的体验非常丝滑。

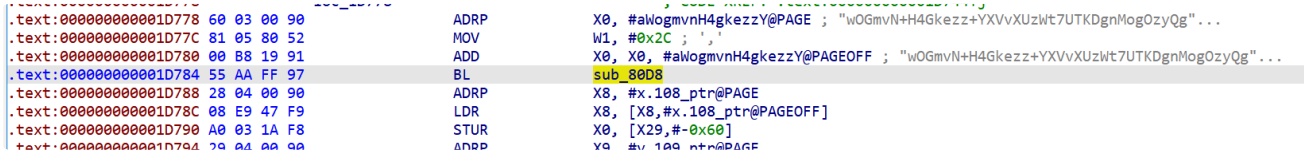
C | 复制代码

```
decrypt:/proc/self/maps
```

接下来我们需要将这种操作对应到所有密文调用上，具体思路是通过 IDAPython 脚本获取解密函数的所有调用地址以及实际参数，然后 Frida Call 主动调用解密函数，最后将明文结果以注释的形式反馈给静态分析。

这里我们在汇编层面做参数的匹配，因为它符合上一篇所说的两点。

1 是部分函数无法 F5，这意味着基于反编译伪代码的匹配会造成遗漏



2 是它的参数只有两个，汇编布局相对简单。跳转的上一条是赋值参数一，再上一条是赋值参数二。

JavaScript | 复制代码

```
.text:000000000000B170 E0 03 00 D0 ADRP X0, #aUwryul
.text:000000000000B174 81 05 80 52 MOV W1, #0x2C ;
.text:000000000000B178 00 88 07 91 ADD X0, X0, #aU
.text:000000000000B17C D7 F3 FF 97 BL sub_80D8
```

观察代码，发现极个别调用的汇编布局比较特殊，我不打算处理这种极端情况，直接作为异常略过。

C | 复制代码

```
X19 在非常上方的位置
.text:0000000000018514 MOV W1, #0x18
.text:0000000000018518 MOV X0, X19
.text:000000000001851C BL sub_80D8
```

下面我们正式开始，首先讨论涉及到的 API 。如果我们想顺着调用地址往上找参数的赋值处，可以使用 `idc.prev_head(ea)` ，它会获取相对于 ea 地址的上一行内容（可以是指令或数据），和它相关的 API 是 `idc.prev_addr(addr)` ，它用于获取上一行指令，这意味着 addr 地址前是数据块的话会跳过它继续往上找。它俩的镜像 API 是 `idc.next_head(ea)` ，`idc.next_addr(addr)` ，寻找方向是向下而非向上。

这里我并不想用这个 API，因为我们观察到汇编布局很简单，参数准备所涉及的指令，其实就三条。假设 ea 是交叉引用的地址，或者说调用地址。 `ea - 12` 和 `ea - 4` 是准备参数 1， `ea - 8` 是准备参数 2。

我们先看参数 2，它很简单，为了避免错误，我们做一下基础的判断，首先判断指令的助记符是否是 `MOV` 。

Python | 复制代码

```
idc.print_insn_mnem(0xB174)
Out[3]: 'MOV'
```

以及指令的第一个操作数是否是 `W1` ，可以使用 `idc.print_operand(ea, n)` ，ea 是指令地址，n 从 0 开始，表示第几个操作数。

```
idc.print_operand(0xB174, 0)
Out[5]: 'W1'
```

在获取操作数 2 这个立即数时，使用这个 API 就不是很方便了，获取到的是文本形式，还得转化成数字。

```
idc.print_operand(0xB174, 1)
Out[6]: '#0x2C'
```

`idc.get_operand_value(ea, n)` 在这种情况下是更好的选择，获取值更方便。

```
idc.get_operand_value(0xB174, 1)
Out[7]: 44
```

读者自己写脚本时，并不总需要找到最合适的 API ， IDA 既没有出版《 IDAPython 编程实践指南》，也没有提供一个良好的 API 文档，因此本着能用就行的原则就行。比如下面两个方式，都可以获取到函数总数，第二种办法更简单，但这个 API 相对生僻，因此第一种在代码里反而更常见。

```
import idaapi
import idautils

len(list(idautils.Functions()))
Out[18]: 276

idaapi.get_func_qty()
Out[19]: 276
```

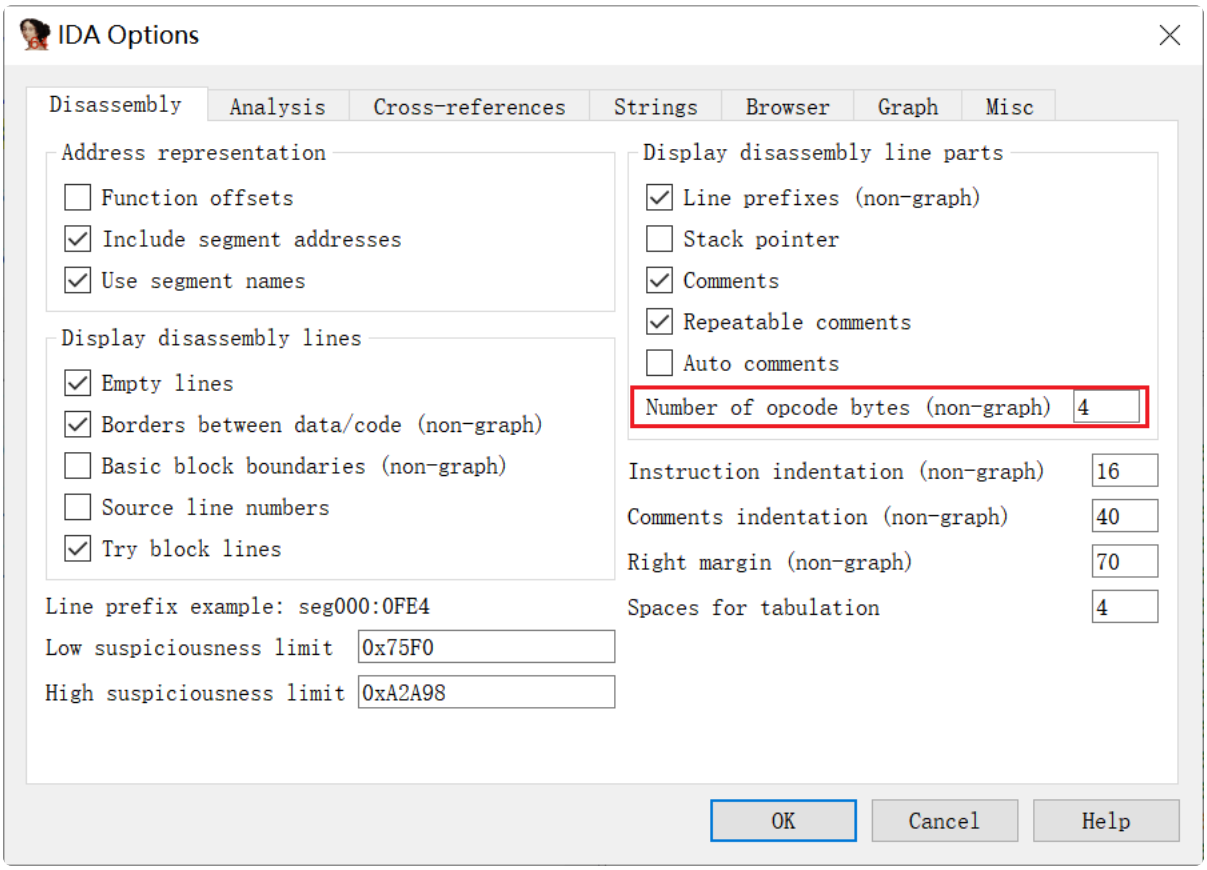
言归正传，目前的代码整合如下

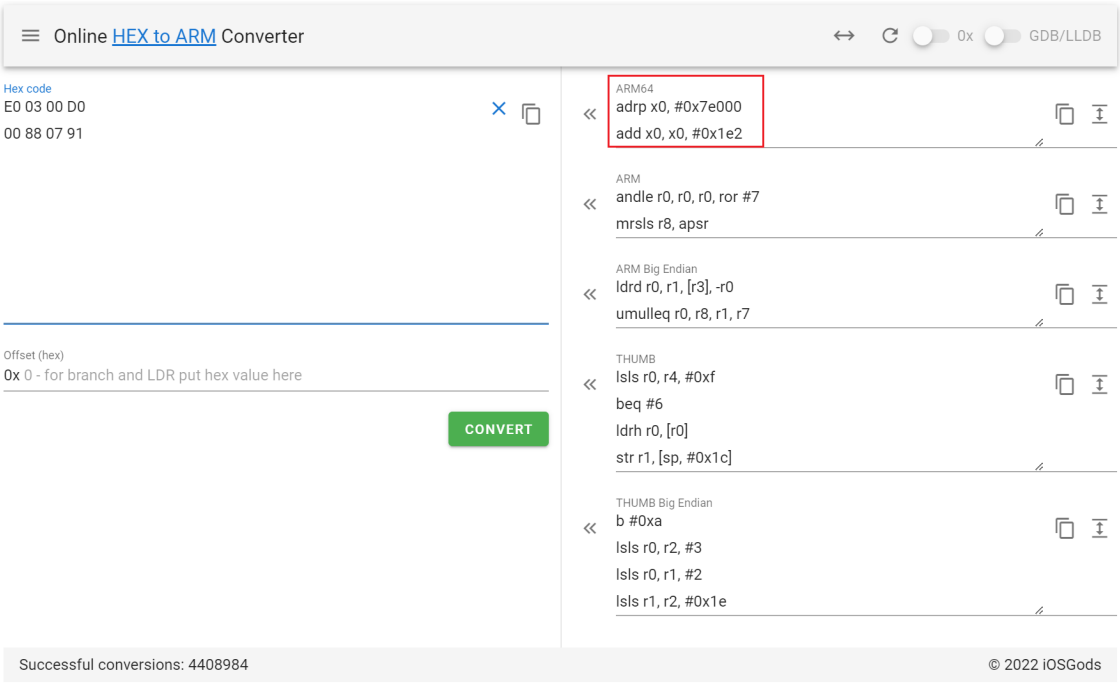
```
import idautils
import idc

xrefs = idautils.CodeRefsTo(0x80D8, 0)
for addr in list(xrefs):
    arg2addr = addr - 8
    if idc.print_insn_mnem(arg2addr) == "MOV" and idc.print_operand(arg2addr, 0
        arg2 = idc.get_operand_value(arg2addr, 1)
        print(arg2)
```

接下来看参数 1 的汇编逻辑，这里就相对复杂一些了，以下面这块为例

```
1  .text:000000000000B170 E0 03 00 D0          ADRP    X0, #aUwryub
2  .text:000000000000B174 81 05 80 52          MOV     W1, #0x2C ;
3  .text:000000000000B178 00 88 07 91          ADD     X0, X0, #aUw
4  .text:000000000000B17C D7 F3 FF 97          BL      sub_80D8
```

IDA 会对汇编代码做优化，让它更方便人类阅读，但这里会给我们带来困扰。armconvert <https://armconverter.com/> 可以让我们看到直白的汇编。将第一条和第三条的机器码放过去看一下，机器码展示通过 Options - General 设置而来。

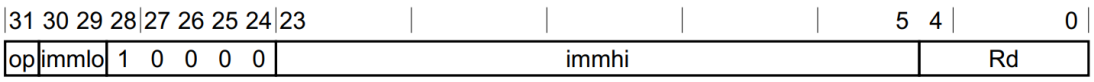




Plain Text | 复制代码

```
1   adrp x0, #0x7e000
2   add x0, x0, #0x1e2
```

我并不想直接讨论 `ADRP` 和 `ADD` 指令，而是从看一下 `ADR` 指令。 `ADR` 和 `ADRP` 是孪生兄弟，如果说名字的相似不足以让你信服，我们再看一下编码规则

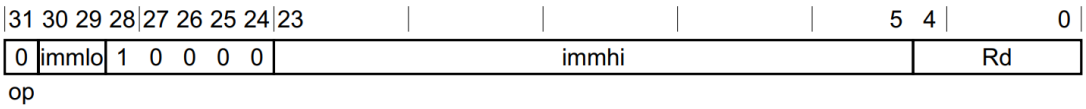


Decode fields	Instruction page
op	
0	ADR
1	ADRP

ADR 还是 ADRP ，仅仅由末尾的一个比特做区分。先看 `ADR` 的原因是什么？我先卖个关子。 `ADR` 是小范围的地址读取指令，它将基于 PC 相对偏移的一个地址值读取到寄存器中。

ADR

Form PC-relative address adds an immediate value to the PC value to form a PC-relative address, and writes the result to the destination register.



Encoding

```
ADR <Xd>, <label>
```

Decode for this encoding

```
integer d = UInt(Rd);
bits(64) imm;

imm = SignExtend(immhi:immlo, 64);
```

大家不用觉得晦涩，我们马上实际操作一下。

假设一条 ADR 指令的机器码是 `CB 00 00 10`，我们该如何解析它？首先按照小端序，将机器码反转成 `10 00 00 CB`，查看其二进制，我这里使用 Windows 自带的计算器。

和 `ADR` 的编码规则进行对应，分成五截：`0 00 10000 0000000000000000110 01011`，解析如下

```
1  immlo = 00
2  immhi = 110
3  Rd = 01011
```

其实这就是手动的反汇编过程，`imm` 是 immediate 立即数的缩写，`lo/hi` 是 low/high 的缩写，连起来解释就是立即数的较低和较高位，因此这个立即数的值是 `0b11000` 也就是 0x18。基于 PC 相对地址，这意味着如果是下面的语境，当前地址或者说 PC 值是 0x35CFC

```
1   .text:0000000000035CFC CB 00 00 10   ADR XXX
```

那么计算地址就是 0x35CFC + 0x18 = 0x35D14。将这个值放到哪个寄存器里？我们还没解析 Rd 呢。`0b01011` 也就是 11，对应于 `X11` 。就下面这样。

```
1   .text:0000000000035CFC CB 00 00 10   ADR X11, 0x35D14
```

编码规定里，这个立即数是有符号类型，可正可负，因此使用 `ADR` 指令可以很方便的获取当前地址上下的某个地址，而且读者可以数一下，在 `ADR` 编码规则里，有 21 比特用于表示这个立即数，去掉符号位还剩下 20 比特，算一下就是可以获取相对于 PC 上下 0xFFFFF 范围的任意地址，用 MB 表示就是 ±1MB。

那么问题来了，如果我们要处理的地址，相对于 PC 地址距离较远呢？上下 1MB 可不是什么大范围，几十几百MB 大小的二进制文件也不少呀。

这时候就自然而然的聊到 `ADRP` 了，它的使用场景就在这里 —— 更大范围的地址读取指令，但我们还是不直接讨论它，而是想一下，如何在单条指令里表示 0 - 0xFFFFFFFF 范围里的任意数，学名即 32 位任意立即数。

在 Intel 那样的变长指令架构中，这不是什么麻烦事。比如下面这条指令，`B8` 是 `mov eax` 的编码规则，后面跟着的四个字节即小端序 0x12345678 。

```
1   0x0000000000000000:  B8 78 56 34 12    mov eax, 0x12345678
```

但 ARM 架构上这事情就难办了。想要表示任意的 32 位立即数，这需要占 4 字节，但 ARM 是定长指令架构，而且定长 4 字节，再加上操作码和其他操作数，完全没法装进去。好比电梯限载四人，现在已经有一人在里面（操作码以及其他操作数），没有办法再进来四个人了。

既然一条指令不行，那么两条指令总行了吧？我想大家也能想到很多办法。

可以一条指令只赋值较高的 16 位，另一条指令只赋值较低的 16 位。在 ARM 中设计了 `MOVW` 和 `MOVT` 指令表示这个逻辑。比如下面两条指令完成了 0xac5a6002 的赋值。

```
1   movw    r0, 0x6002
2   movt    r0, 0xac5a
```

可以一条指令是内存访问指令，另外四字节是对应的数。 `LDR` + `Data` 是各种 inline hook 框架最喜欢的 32位无条件跳转方案。

```
LDR PC, [PC, #imm]
……
addr
```

`ADRP` 其实有点像第一种方案，首先它在编码上和 ADR 几乎没差别，立即数的实际表示范围是 ±0xFFFFF，但是它将这个数值乘上 0x1000，那么表示范围就成了 -0xFFFFF000 ~ 0xFFFFF000 范围内每隔 0x1000 距离的数。表示的范围变大了，但表示的能力稀疏了。

回到我们的样本，进行实践

```
.text:000000000000B170 E0 03 00 D0                 ADRP            X0, #aUwryul
.text:000000000000B174 81 05 80 52                 MOV             W1, #0x2C ;
.text:000000000000B178 00 88 07 91                 ADD             X0, X0, #aU
.text:000000000000B17C D7 F3 FF 97                 BL              sub_80D8
```

E0 03 00 D0 小端序翻转： `1 10 10000000000000000011111 00000`

```
immlo = 10
immhi = 11111
Rd = 0
```

立即数是 `0b1111110` 即 0x7E，寄存器是 `X0` 。 `0x7E * 0x1000 = 0x7E000` 。PC 是 0xB170，向下对 0x1000 取整（“基于 PC 相对偏移”这条规则会和相应的和 0x1000 做对齐）是 0xB000，即 `ADRP X0,0x89000` 。读者可能会注意到，ArmConvert 中翻译为 `adrp x0, #0x7e000` ，这是因为它默认 PC 是 0。

我们想取的地址并不总能恰好和 0x1000 对齐，所以还需要配合一条 `ADD` 或 `SUB` 指令，加上或减去 0x1000 范围内的某个值。 `00 88 07 91` 用 ArmConvert 看到是 `add x0, x0, #0x1e2` ，0x89000 + 0x1e2 = 0x891E2。IDA 中跳过去看看，完美对上。

```
.rodata:00000000000891CB 67 61 6E 69+                   ; sub_80D8+CF4↑o
.rodata:00000000000891E2 55 57 52 59+aUwryubyl2xxvag DCB "UWRYubYl2XXvaG3S9r5ezWcxX/VsRigluNW58+nIYq4=",0
.rodata:00000000000891E2 75 62 59 6C+                   ; DATA XREF: sub_9E2C+1080↑o
.rodata:00000000000891E2 32 58 58 76+                   ; sub_9E2C+1088↑o ...
```

大家可能会烦恼，IDA 对汇编所作的自动解析，似乎让我们没法直接解析这个地址值。不必担心， `get_operand_value` API 还是可以正常使用。以下面的代码片段为例

JavaScript 复制代码

```
.text:000000000000B170 E0 03 00 D0                ADRP            X0, #aUwryub
.text:000000000000B174 81 05 80 52                MOV             W1, #0x2C ;
.text:000000000000B178 00 88 07 91                ADD             X0, X0, #aUw
.text:000000000000B17C D7 F3 FF 97                BL              sub_80D8
```

C 复制代码

```
hex(idc.get_operand_value(0xB170, 1))
Out[12]: '0x89000'

hex(idc.get_operand_value(0xB178, 2))
Out[13]: '0x1e2'
```

下面做代码的整合

C 复制代码

```
import ida_bytes
import idautils
import idc

xrefs = idautils.CodeRefsTo(0x80D8, 0)
record = []
for addr in list(xrefs):
    try:
        arg2addr = addr - 8
        arg1part1 = addr - 12
        arg1part2 = addr - 4
        if idc.print_insn_mnem(arg2addr) == "MOV" and idc.print_operand(arg2ad
            arg2 = idc.get_operand_value(arg2addr, 1)
            arg1 = idc.get_operand_value(arg1part1, 1) + idc.get_operand_value
            encryptText = ida_bytes.get_strlit_contents(arg1, -1, -1).decode('
            record.append([addr, encryptText, arg2])
    except:
        pass

print(record)
```

收集结果如下

C 复制代码

```
[[42920, 'X2WZEOYLQ51XAqWG4e50TA==', 24], [43112, 'X2WZEOYLQ51XAqWG4e50TA==', 2
```

Frida 做主动调用

JavaScript | 复制代码

```
// 获取解密函数绝对地址
var base_addr = Module.findBaseAddress("libsmsdk.so");
var decrypt_address = base_addr.add(0x80d8)
// 根据解密函数的定义构造它
var decrypt_Fun = new NativeFunction(decrypt_address, "pointer", ["pointer", '

// 封装对解密函数的调用，并获取返回值
function decrypt(encryptContent, length){
    var encryptAddress = Memory.allocUtf8String(encryptContent);
    var decryptAddress = decrypt_Fun(encryptAddress, length);
    return decryptAddress.readUtf8String();
}

var decryptList = [[42920, 'X2WZEOYLQ51XAqWG4e50TA==', 24], [43112, 'X2WZEOYL(

for (var i=0; i< decryptList.length; i++)
{
    var one = decryptList[i]
    var decryptContent = decrypt(one[1], one[2]);
    console.log(one[0]+" "+decryptContent)
}

```

结果如下，左侧是十进制表示的调用地址，右侧是解密字符串。

Shell | 复制代码

```
42920 r-xp
43112 r-xp
44728 /proc/self/maps
44756 /proc/self/maps
45408 /proc/self/maps
45436 /proc/self/maps
46664 libc.so
46692 libc.so
57372 substrate
57456 substrate
57832 XposedBridge.jar
58080 XposedBridge.jar
61080 /proc/self/cmdline
61312 /proc/self/cmdline
63588 /proc/net/arp
63616 /proc/net/arp
69784 tun
69988 tun
70972 wlan
71024 wlan
72816 tun
72980 tun
101820 /system/xbin/su
102196 /system/xbin/su
105948 arm64-v8a
106212 arm64-v8a
106424 arm
106672 arm
106904 x86_64
107132 x86_64
108064 arm64-v8a
108252 arm64-v8a
108732 mips
108816 mips
109272 x86
109356 x86
120708 /proc/asound/pcm
120852 /proc/asound/pcm
121368 /proc/asound/pcm
121412 /proc/asound/pcm
132812 com/ishumei/dfp/SMSDK
132912 com/ishumei/dfp/SMSDK
137516 com/ishumei/dfp/SMSDK
137716 com/ishumei/dfp/SMSDK
152832 substrate
153064 substrate
153264 /system/fonts
153460 /system/fonts
153636 /data/system
```

```
153840 /data/system
154016 xposed
154252 xposed
154464 /data/system
154664 /data/system
154848 /data/system
155056 /data/system
155244 /system/framework
155452 /system/framework
155644 /system/fonts
155840 /system/fonts
156364 /system/bin
156408 /system/bin
160020 /data/system
160064 /data/system
160760 /vendor/lib
160804 /vendor/lib
163452 /vendor/firmware
163496 /vendor/firmware
167040 /system/bin/ls
167108 /system/bin/ls
```

读者可以自行测试，以汇编注释的形式把明文注释到交叉引用地址上。

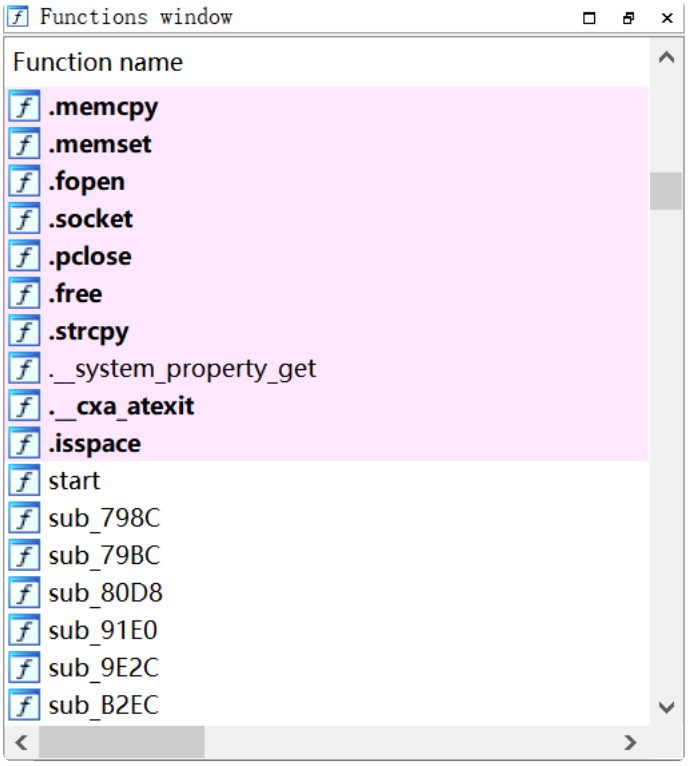
这里再讲一个 API，有时候很有用。`idaapi.get_arg_addrs(ea)`，其中 `ea` 是发起函数调用的指令地址，它会读取这个函数调用所对应的，每个参数初始化的汇编地址。听着有些拗口，举个例子。依然是是下面这个熟悉的汇编代码片段。如果 `ea` 设置为 0xB17C

Shell | 复制代码

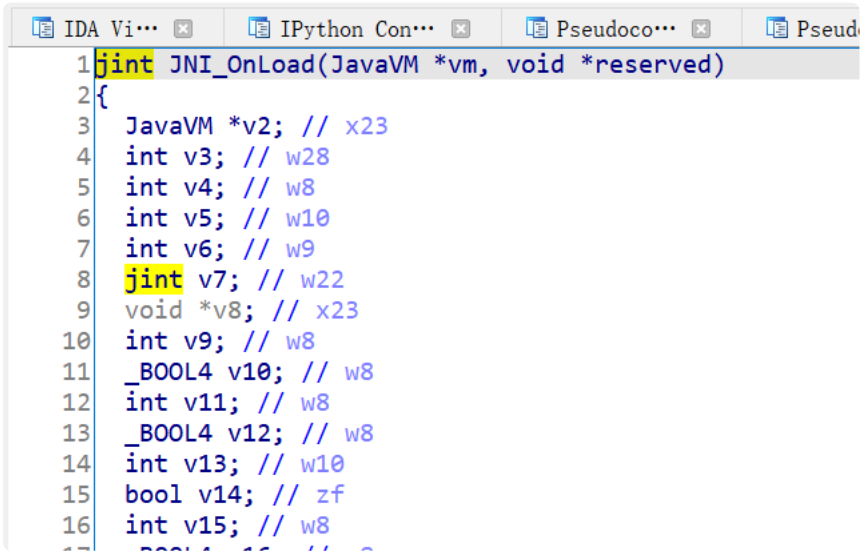
```
.text:000000000000B170 E0 03 00 D0                ADRP            X0, #aUwryub
.text:000000000000B174 81 05 80 52                MOV             W1, #0x2C ;
.text:000000000000B178 00 88 07 91                ADD             X0, X0, #aUw
.text:000000000000B17C D7 F3 FF 97                BL              sub_80D8
```

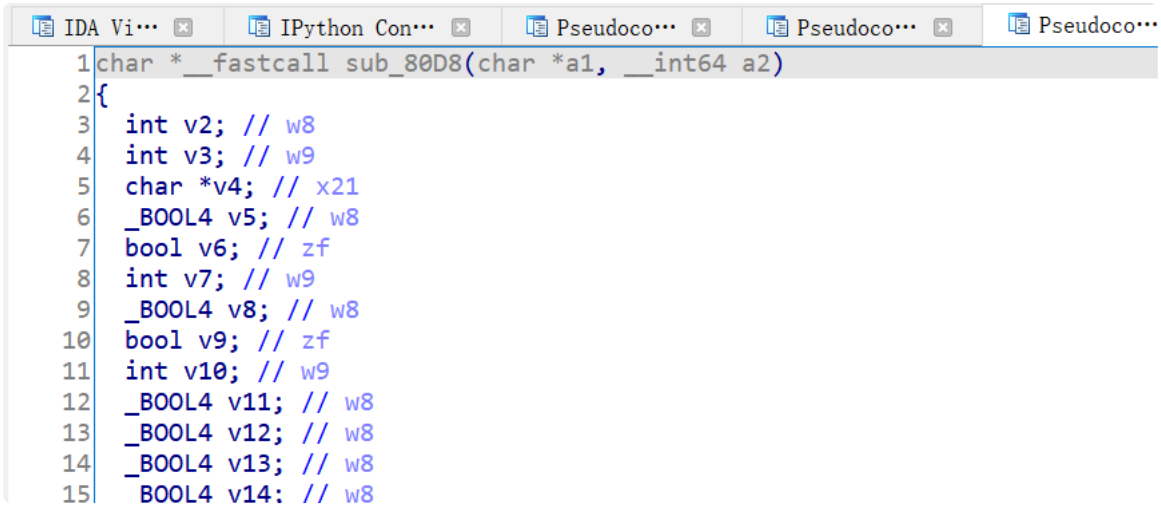
首先，IDA 会根据函数定义确定参数个数，此处函数调用指向 `sub_80D8`，它有两个参数。然后根据当前函数所采用的函数调用约定，确定这些参数如何传递给 `sub_80D8`，ARM64 默认采用 ATPCS 调用约定，前八个参数通过 X0 - X7 或 W0 - W7 传递，多于 8 个的参数通过堆栈传递，因此 `sub_80D8` 的两个参数对应于 `X/W0` 和 `X/W1`。IDA 会找到这俩寄存器在函数调用前，最后的赋值地址。

应该说，在函数存在大量参数或汇编布局复杂的情况中，这个 API 确实能优化对参数的匹配。下面我们实际测试一下。首先我们得做一个处理——帮助IDA 明确函数参数。这是什么意思呢？读者可能会注意到，下面函数列表包含的各种函数，在显示上并不相同，部分字体会加黑。



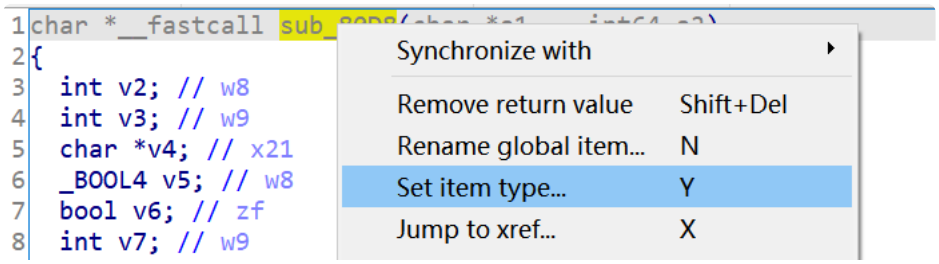
反编译界面的函数声明也与之对应有深有浅。





浅色的函数，其参数个数和类型是 IDA 自行推断出来的，可能并不准确，深色的函数，其参数准确，往往是已识别的库函数或 `JNI_OnLoad` 这样的主要函数。

`idaapi.get_arg_addrs` 并不相信自己推断出来的函数类型，需要我们的辅助，给函数重新做定义。



这件事并不复杂，Set item type 然后修改，这里不必做修改，因为 IDA 的分析没问题，所以直接点确定即可。IDA 认为我们修改或确认过的函数定义是准确的，这个函数随之变成深色，可以正常使用 `idaapi.get_arg_addrs` 了。

Shell | 复制代码

```
idaapi.get_arg_addrs(0xB17C)
Out[23]: [45432, 45428]
```

读者千万不要误认为这个 API 可以直接获取函数的实际参数，其作用和原理在上面已经讲的很清楚了。这里总结一下本节的处理思路

- 找到解密函数
- 分析汇编代码匹配和计算出实际参数
- 使用 Frida 主动调用获得结果
- 以注释或某种形式增强静态分析体验

三、Unidbg Call 解密函数

样本：libanogs.7z (1.3 MB)

交叉引用量位列第三的 sub_27DDC0 是字符串明文获取函数，这个样本我们换个花样，采用 Unidbg 来做字符串的解密处理。

首先在解密函数的入参上，和其他样本很不一样，它只有一个参数，看起来是小整数，返回指向明文的指针。

C | 复制代码

```
v7 = sub_27DDC0(4732);
v8 = sub_27DDC0(13295);
```

使用 Unidbg Call

Java | 复制代码

```java
package com.angos;

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.dvm.DalvikModule;
import com.github.unidbg.linux.android.dvm.VM;
import com.github.unidbg.memory.Memory;
import com.github.unidbg.pointer.UnidbgPointer;

import java.io.File;
import java.io.IOException;

public class Angos {
    private final AndroidEmulator emulator;
    private final VM vm;
    private final Module module;

    public Angos() {
        emulator = AndroidEmulatorBuilder.for64Bit().build();
        Memory memory = emulator.getMemory();
        memory.setLibraryResolver(new AndroidResolver(23));
        vm = emulator.createDalvikVM();
        DalvikModule dm = vm.loadLibrary(new File("unidbg-android/src/test/
        module = dm.getModule();
    }

    public void destroy() throws IOException {
        emulator.close();
    }

    public void callDecrypt(int num){
        Number number = module.callFunction(emulator, 0x27ddc0, num);
        UnidbgPointer ret = UnidbgPointer.pointer(emulator, number);
        System.out.println("decrypt address:"+ret+" str:"+ret.getString(0))
    }

    public static void main(String[] args) {
        Angos angos = new Angos();
        angos.callDecrypt(13295);

        try {
            angos.destroy();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

```

结果如下，效果很好。

```
decrypt address:RW@0x404953d5[libanogs.so]0x4953d5 str:func=WB_GetTPShellVersi
```

接下来可以如法炮制，首先在 IDA 中找到所有的交叉引用位置以及参数，我使用 `idaapi.get_arg_addrs(ea)` 简化我们的任务。

```python
import idaapi
import idautils
import idc

xrefs = idautils.CodeRefsTo(0x27DDC0, 0)
for addr in list(xrefs):
    try:
        numAddress = idaapi.get_arg_addrs(addr)[0]
        num = idc.get_operand_value(numAddress, 1)
        print("call address:"+hex(addr) + " num:" + hex(num))
    except:
        pass

```

代码简洁的过分，而且反馈正常。

```
call address:0x258c2c num:0x4964
call address:0x258ddc num:0x4979
call address:0x258f84 num:0x498e
call address:0x25a490 num:0x49a7
call address:0x25a654 num:0x49b9
```

通过列表或某种容器传给 Unidbg callDecrypt 调用，打印调用地址与解密明文，最后用汇编注释进行备注即可。请读者自行尝试。

有人可能会对这个解密算法的具体机制产生兴趣。我们使用 Unidbg 的 traceWrite 探究一下，在 Unidbg 里分析算法的体验实在是太好了。

我首先在解密结果后下了一个断点

```java
public void callDecrypt(int num){
    Number number = module.callFunction(emulator, 0x27ddc0, num);
    UnidbgPointer ret = UnidbgPointer.pointer(emulator, number);
    System.out.println("decrypt address:"+ret+" str:"+ret.getString(0));
    emulator.attach().debug();
}
```

断下来后看一下 Unidbg 中的内存布局

```
vm
[  0][    libdl.so] [0x405e0000-0x40600000]libdl.so
[  1][     libc.so] [0x40600000-0x406f0000]libc.so
[  2][     libm.so] [0x406f0000-0x40740000]libm.so
[  3][   libc++.so] [0x404e0000-0x405e0000]libc++.so
[  4][   liblog.so] [0x404c0000-0x404e0000]liblog.so
[  5][libstdc++.so] [0x40740000-0x40760000]libstdc++.so
[  6][ libanogs.so] [0x40000000-0x404c0000]libanogs.so
```

在不额外干涉 SO 加载流程的情况下，目标 SO 的基地址总是 0x40000000，我们注意到解密字符串的地址 0x404953d5 仍然在 SO 的范围内，IDA 中跳过去看，发现在 bss 段。

```
.bss:00000000004953D5                % 1
.bss:00000000004953D6                % 1
.bss:00000000004953D7                % 1
.bss:00000000004953D8                % 1
.bss:00000000004953D9                % 1
.bss:00000000004953DA                % 1
.bss:00000000004953DB                % 1
.bss:00000000004953DC                % 1
.bss:00000000004953DD                % 1
.bss:00000000004953DE                % 1
```

感兴趣的读者可以使用 Unidbg 对解密算法做还原，尽管存在恼人的指令替换和控制流平坦化，但细心分析后会发现，它本质上是异或解密。

四、新与旧

我们常常会产生一种感觉，似乎技术发展一日千里，几年前的项目和思路早已不值得学习，而自己，是追逐着最新技术的富有经验的猎手。从各种意义上说，这可能是一种错觉。

在绝大多数情况下，我们不是在吃 WIndows 对抗的剩饭，就是在吃大公司、国外同行，以及开发领域的剩饭，有时候甚至饭都馊了也没吃上。在这一节里我们看一下 2012 年 IDA 插件大赛的一个

参赛作品，它主要的用途也是字符串解密，和现在我们所作的事没有差别。

代码下载：Krypton_2012_Hex-Rays_Contest.zip (491 KB)

下面对这个项目的主要部分进行阐述和讨论，因为年代久远， Krypton 是用 Python 2 写的，而且使用的 API 对应于 IDA 5.X 版本，我们没法在当前版本做使用上的复现。

在插件的初始化函数中对架构做了判断，只适配了 X86，这是一碗 PC 的剩饭。

Python | 复制代码

```python
def init(self):
    # 只处理 X86，其余架构不支持
    if getProcessorName().startswith('metapc'):
        return PLUGIN_KEEP
    return PLUGIN_SKIP
```

在插件的具体逻辑里，首先它会列出交叉引用被使用最多的 25 个函数。

Python | 复制代码

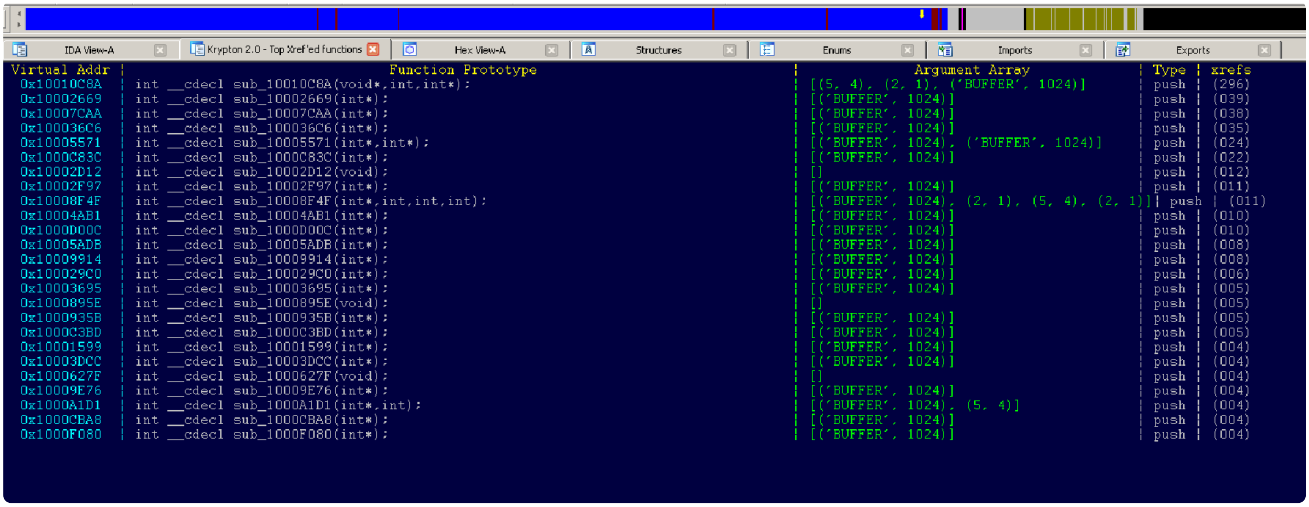
```python
def __init__(self, topN=25):
    self.openLogFile("ProtoAnalyser")
    # 获取函数总数
    self.no_of_funcs = idaapi.get_func_qty()
    # topN = 25,如果函数总数少于 25，就全部列出来
    if topN > self.no_of_funcs:
        self.topNum = self.no_of_funcs
    else:
        self.topNum = topN
        self.allXrefFuncs = self.getTopXrefFunctions()
```

Python | 复制代码

```python
def getTopXrefFunctions(self):
    funcs = []
    for fnum in xrange(self.no_of_funcs):
        f_ea = idaapi.getn_func(fnum).startEA
        if f_ea != BADADDR:
            # 统计交叉引用被引用数
            funcs.append((GetFunctionName(f_ea), f_ea, len(self.getXrefs(f_ea
            # 排序
            funcs = sorted(funcs, key=lambda x: x[2], reverse=True)
            return funcs
```

这么做是有效的，前面我们讨论函数范畴时也讨论过这一点。一般来说，解密函数会被很多地方调用很多次，这会导致它的交叉引用量比一般函数高出很多，所以字符串解密函数往往在最多交叉引用的那一批函数里。即使不在里面，我们分析这些在程序中出现最多次的函数也是有益处的，可以增进对程序整体的理解。

找到解密函数后，`ProtoAnalyser` 是其中重要的类。我们需要帮助 Krypton 明确解密函数是哪一个，以及它的形参和实参，然后 Krypton 会根据 X86 所采用的函数调用规则，在汇编代码中找到参数的最后赋值位置，以及尽力解析具体的值。



读者可能会意识到，这就是“分析汇编代码匹配和计算出实际参数”的步骤。那么下一步 Frida Call 呢？这个项目会如何处理？IDA 在调试器状态下，提供了名为 AppCall 的特性，用于实现函数的主动调用。Krypton 的 DebuggerCtrl 代码逻辑让样本进入符合需求的调试状态，然后使用 IDA AppCall 对解密函数做主动调用。

AppCall 的整体功能和 Frida Call 是一致的，但它依赖于 IDA Debugger ，容易被反调试。除此之外，在易用性和通用性上 AppCall 比 Frida Call 差很多，所以现在用的人不多，对它感兴趣的话可以看 《IDA Pro 权威指南》的 26.3 小节做更多了解。

最后，Krypton 将解密后的明文以汇编注释的形式添加在调用位置上，辅助做后续的静态分析。

Python | 复制代码

```python
def write2IDB(self):
    global decResults
    try:
        if self.decfn.getEA() in decResults:
            (ea,x) = decResults[self.decfn.getEA()]
            for (k,v) in x.iteritems():
                MakeRptCmt(k, v)
            kprint("Decrypted strings Written to IDB at respective references
    except Exception,e:
        kprint("Exception!! @decViewer:write2IDB, ", e)
    return True
```

应该说，整体的思路、流程、方法和本篇所讲的内容是完全一致的，而且还提供了很好的 GUI 界面。它用 AppCall 做函数调用，我们用 Frida Call 做调用，就这么点区别而已。很多时候，我们只是新瓶装旧酒，只不过新瓶子确实更漂亮好用一些，但本质并没有什么差异。

五、总结

简单来说，第二篇侧重于复写解密函数 + 在反编译代码中匹配参数，第三篇侧重于 Frida/Unidbg 调用解密函数 + 在汇编中匹配参数。除此之外， 第三篇重点阐述了一个问题 —— 老项目并不总陈旧无用，新项目也可能是新瓶装旧酒。