四、初探脚本开发——IDA结合Frida（一）

一、前言

大家一定都用过 IDA 静态分析 + Frida Hook 这套联合调试方案，或许我们可以用 IDA 脚本让两者的交互体验更丝滑一些?

二、需求分析

回想一下 Frida 的使用习惯，在分析 Native 我们主要有这么几种需求

函数 Hook，如果是有符号函数，比如导出函数，那么既可以计算绝对地址进行 Hook ，也可以用 `findExportByName` 直接获取地址。举例如下

来自sakura的代码片段 JavaScript | 复制代码

```
var sakura_add_addr1 = Module.findExportByName("libnative-lib.so", "_Z10sakura_addii");
var sakura_add_addr2 = libnative_lib_addr.add(0x0F56C) ;
```

如果是无符号函数，就只能通过绝对地址进行 Hook 。我们是否可以一键生成函数对应的 Hook 脚本？我想当然可以，比如像下面这样，Hook 函数，打印参数和返回值。

JavaScript | 复制代码

```
function hook_sub_A701C(){
    var base_addr = Module.findBaseAddress("libtiger_tally.so");

    Interceptor.attach(base_addr.add(0xa701c), {
        onEnter(args) {
            console.log("call sub_A701C");
            console.log("arg0:"+args[0]);
            console.log("arg1:"+args[1]);

        },
        onLeave(retval) {
            console.log(retval);
            console.log("leave sub_A701C");
        }
    });
}
```

为了简单起见，不管函数是否有符号，我们都采用地址方式做 Hook ，这样只需要一个模板。函数 Hook 是最常见的一类需求，第二类需求是对函数内某处地址做 Hook ，并查看寄存器值和上下文，我们一般称这种需求为 inline hook ，代码大概长下面这样。

JavaScript | 复制代码

```
function hook_0x10cb30(){
    var base_addr = Module.findBaseAddress("libtiger_tally.so");

    Interceptor.attach(base_addr.add(0x10cb30), {
        onEnter(args) {
            console.log("call 0x10cb30");
            console.log(JSON.stringify(this.context));
        },
    });
}
```

三、动手实践

首先确认模板，下面是函数 Hook 模板

JavaScript | 复制代码

```javascript
function hook_$functionName(){
  var base_addr = Module.findBaseAddress("$soName");

  Interceptor.attach(base_addr.add($offset), {
    onEnter(args) {
      console.log("call $functionName");
      // 简单打印入参
      console.log("arg0:"+args[0]);
      console.log("arg1:"+args[1]);

    },
    onLeave(retval) {
      // 如果有返回值，则打印返回值
      console.log(retval);
      console.log("leave $functionName");
    }
  });
}
```

从模板中可见，我们需要通过 IDA 获取函数名、动态库文件名、函数偏移地址、函数参数和返回值情况。其次是 inline Hook 模板

JavaScript | 复制代码

```javascript
function hook_$offset(){
    var base_addr = Module.findBaseAddress("$soName");

    Interceptor.attach(base_addr.add($offset), {
        onEnter(args) {
            console.log("call $offset");
            console.log(JSON.stringify(this.context));
        },
    });
}
```

inline hook 模板相对简单，只需要关注动态库文件名、函数偏移地址。

下面书写简单的实现代码

Python | 复制代码

```python
from string import Template

hook_function_template = """
function hook_$functionName(){
    var base_addr = Module.findBaseAddress("$soName");

    Interceptor.attach(base_addr.add($offset), {
        onEnter(args) {
            console.log("call $functionName");
            $args
        },
        onLeave(retval) {
            $result
            console.log("leave $functionName");
        }
    });
}
"""

inline_hook_template = """
function hook_$offset(){
    var base_addr = Module.findBaseAddress("$soName");

    Interceptor.attach(base_addr.add($offset), {
        onEnter(args) {
            console.log("call $offset");
            console.log(JSON.stringify(this.context));
        },
    });
}
"""

logTemplate = 'console.log("arg$index:"+args[$index]);\n'


def generate_printArgs(argNum):
    if argNum == 0:
        return "// no args"
    else:
        temp = Template(logTemplate)
        logText = ""
        for i in range(argNum):
            logText += temp.substitute({'index': i})
            logText += "            "
        return logText


def generate_for_func(soName, functionName, offset, argNum, hasReturn):
    # 根据参数个数打印
    argsPrint = generate_printArgs(argNum)
    # 根据是否有返回值判断是否打印retval
    retPrint = "// no return"
    if hasReturn:
        retPrint = "console.log(retval);"
    # 使用Python提供的Template字符串模板方法
    temp = Template(hook_function_template)
    result = temp.substitute({'soName':soName, "functionName":functionName, "offset":hex(offset), "args": argsPrint, "resul
    print(result)


def generate_for_inline(soName, offset):
    temp = Template(inline_hook_template)
    result = temp.substitute({'soName': soName, "offset": hex(offset)})
    print(result)

```

假设需要Hook libnative-lib.so 中的 sub_3DD50 函数，反编译代码如下

JavaScript 复制代码

```
int __fastcall sub_3DD50(int a1, int a2, int a3)
{
  void *v6; // r4

  v6 = malloc(0xB0u);
  sub_3A8F8(a1, v6, 10, 4);
  sub_3CC68(a2, a3, v6, 10);
  return sub_64F30(v6);
}
```

动态库文件名是 libnative-lib.so；

函数名是 sub_3DD50；

函数偏移地址，如果是 ARM64 ，那就是 0x3DD50 ，如果是 ARM32 下的 thumb 模式，地址需要+1；

参数是三个参数，存在返回值。

传入函数如下

Python 复制代码

```
generate_for_func("libnative-lib.so", "sub_3DD50", 0x3dd51, 3, True)
```

打印效果如下，应该说还挺不错的。

JavaScript 复制代码

```
function hook_sub_3DD50(){
    var base_addr = Module.findBaseAddress("libpdd_secure.so");

    Interceptor.attach(base_addr.add(0x3dd51), {
        onEnter(args) {
            console.log("call sub_3DD50");
            console.log("arg0:"+args[0]);
            console.log("arg1:"+args[1]);
            console.log("arg2:"+args[2]);

        },
        onLeave(retval) {
            console.log(retval);
            console.log("leave sub_3DD50");
        }
    });
}
```

我们已经确定了想要的效果，该怎样才能在 IDA 中获取这些信息呢？需要注意，下面对各种 API 的测试与展示均通过 IPyIDA 插件完成，它很优雅，我们前面介绍过它。

IDA 提供了获取文件名的 API

PowerShell 复制代码

```
import idaapi
idaapi.get_root_filename()
Out[1]: 'libpdd_secure.so'
```

需要注意，在编写 IDAPython 脚本时，我们常常会遇到一个问题——找不到所求功能最精准对应的那个API，往往通过其他方式间接实现功能。比如获取文件名这件事，有时也用下面这个 API

PowerShell 复制代码

```
import os

os.path.basename(idaapi.get_input_file_path())
Out[63]: 'libpdd_secure.so'
```

这也给大家提供了一种启示，即我文章演示中所采用的API，未必是最简单或最有效的。读者可以在自己的学习和复现过程中探索更好的思路和API。

回归正轨，在IDA中如何获得具体函数的信息呢？当我们在IDA中指定某一个函数或地址时，该如何获取它或它所属的函数名，参数个数，是否存在返回值这些信息呢？

首先是函数名的获取

PowerShell 复制代码

```
idaapi.get_func_name(0x3dd50)
Out[2]: 'sub_3DD50'
```

入参为函数体内任意地址，返回值是所属函数的函数名。如果函数本身是有符号的，返回自身名字，如果本身无符号，则返回 IDA 所进行的自动命名，大多数情况下都是SUB_Address，意为首地址为 Address 的子程序。

那么参数个数和返回值是否是 void ，如何获取呢？我们看反编译代码才确认这些信息，比如下面这样的 F5 结果

C 复制代码

```
int __fastcall sub_3DD50(int a1, int a2, int a3)
{
    void *v6; // r4

    v6 = malloc(0xB0u);
    sub_3A8F8(a1, v6, 10, 4);
    sub_3CC68(a2, a3, v6, 10);
    return sub_64F30(v6);
}
```

因此我们需要使用 IDAPython 的反编译相关 API ，查看反编译所得伪代码。

Python 复制代码

```
cfun = idaapi.decompile(0x51578)
```

对参数个数的查询

Python 复制代码

```
cfun.arguments
Out[89]:
[<ida_hexrays.lvar_t; proxy of <Swig Object of type 'lvar_t *' at 0x0000021C95A62D50> >,
 <ida_hexrays.lvar_t; proxy of <Swig Object of type 'lvar_t *' at 0x0000021C95A62780> >,
 <ida_hexrays.lvar_t; proxy of <Swig Object of type 'lvar_t *' at 0x0000021C95A62420> >,
 <ida_hexrays.lvar_t; proxy of <Swig Object of type 'lvar_t *' at 0x0000021C95A62DB0> >]

len(cfun.arguments)
Out[90]: 4
```

返回值是否是 void ？我没找到直接对应的API，尽管理论上应该有，读者找到也可以告知我。我选择解析函数原型，根据是否以 void 开头来判断是否存在返回值，同时得避免误识别 void * 。众所周知，void * 表示返回任意类型的值的指针。

IPyIDA Python 复制代码

```
dcl = cfun.print_dcl()

print(dcl)
SOHETBint __fastcall sub_4C228SOH (STX _DWORD *a1SOH ,STX int a2SOH ,STX int a3SOH ,STX int a4SOH )STX STXETB

cfun.print_dcl()
Out[99]: '\x01\x17int __fastcall sub_4C228\x01\t(\x02\t_DWORD *a1\x01\t,\x02\t int a2\x01\t,\x02\t int a3\x01\t,\x02\t int a
```

`print_dcl` API 会返回函数原型，读者可能会期待它返回下面这样

`int __fastcall sub_4C228(_DWORD *a1, int a2, int a3, int a4)`

但在 IPyIDA 中对这个 API 测试打印时，返回的内容中似乎穿插了一些奇怪的字节。不要觉得意外，应该将打印的反编译代码理解为一种富文本。

IDA View-A | Pseudocode-A | Hex View-1

```
1 int __fastcall sub_4C228(_DWORD *a1, int a2, int a3, int a4)
2 {
3   int v4; // lr
4   int v5; // r12
5   int v6; // r0
6   int v7; // r1
7   int v8; // r3
8   int v9; // r1
9   int v10; // r1
10  int v11; // r1
11  int v12; // r1
12  int v13; // r1
13  int v14; // r1
14  int v15; // r1
15  int v16; // r0
```

请读者仔细观察反编译得到的伪代码，不同文本部分存在各种颜色变化。通过 `print_dcl` API 打印出的函数原型，其中的奇怪字节就是颜色相关的信息。我们需要纯文本，可以通过 API 去除这些颜色信息。

IPyIDA Python 复制代码

```
import ida_lines

ida_lines.tag_remove(cfun.print_dcl())
Out[103]: 'int __fastcall sub_4C228(_DWORD *a1, int a2, int a3, int a4)'
```

API 组合在一起，代码如下

Python 复制代码

```
import ida_lines
import idaapi


def get_argnum_and_ret(address):
    cfun = idaapi.decompile(address)
    argnum = len(cfun.arguments)
    ret = True
    dcl = ida_lines.tag_remove(cfun.print_dcl())
    if (dcl.startswith("void ") is True) & (dcl.startswith("void *") is False):
        ret = False
    return argnum, ret
```

在 IPyIDA 中做测试

IPython Console

```
In [104]: import ida_lines
     ...: import idaapi
     ...:
     ...:
     ...: def get_argnum_and_ret(address):
     ...:     cfun = idaapi.decompile(address)
     ...:     argnum = len(cfun.arguments)
     ...:     ret = True
     ...:     dcl = ida_lines.tag_remove(cfun.print_dcl())
     ...:     if (dcl.startswith("void ") is True) & (dcl.startswith("void *") is False):
     ...:         ret = False
     ...:     return argnum, ret
     ...:

In [105]: get_argnum_and_ret(0x12790)
Out[105]: (3, True)

In [106]: get_argnum_and_ret(0x2bd08)
Out[106]: (2, False)

In [107]:
```

在我的样本上反馈正常。接下来组合功能，代码如下

Python | 复制代码

```python
from string import Template
import ida_lines
import idaapi
import idc

hook_function_template = """
function hook_$functionName(){
    var base_addr = Module.findBaseAddress("$soName");

    Interceptor.attach(base_addr.add($offset), {
        onEnter(args) {
            console.log("call $functionName");
            $args
        },
        onLeave(retval) {
            $result
            console.log("leave $functionName");
        }
    });
}
"""
inline_hook_template = """
function hook_$offset(){
    var base_addr = Module.findBaseAddress("$soName");

    Interceptor.attach(base_addr.add($offset), {
        onEnter(args) {
            console.log("call $offset");
            console.log(JSON.stringify(this.context));
        },
    });
}
"""

logTemplate = 'console.log("arg$index:"+args[$index]);\n'


def generate_printArgs(argNum):
    if argNum == 0:
        return "// no args"
    else:
        temp = Template(logTemplate)
        logText = ""
        for i in range(argNum):
            logText += temp.substitute({'index': i})
            logText += "            "
        return logText


def generate_for_func(soName, functionName, offset, argNum, hasReturn):
    # 根据参数个数打印
    argsPrint = generate_printArgs(argNum)
    # 根据是否有返回值判断是否打印retval
    retPrint = "// no return"
    if hasReturn:
        retPrint = "console.log(retval);"
    # 使用Python提供的Template字符串模板方法
    temp = Template(hook_function_template)
    result = temp.substitute(
        {'soName': soName, "functionName": functionName, "offset": hex(offset), "args": argsPrint, "result": retPrint})
    print(result)


def generate_for_inline(soName, offset):
    temp = Template(inline_hook_template)
    result = temp.substitute({'soName': soName, "offset": hex(offset)})
    print(result)


def get_argnum_and_ret(address):
    cfun = idaapi.decompile(address)
    argnum = len(cfun.arguments)
    ret = True
```

```
    dcl = ida_lines.tag_remove(cfun.print_dcl())
    if (dcl.startswith("void ") is True) & (dcl.startswith("void *") is False):
        ret = False
    return argnum, ret


def generate_for_func_by_address(addr):
    soName = idaapi.get_root_filename()
    functionName = idaapi.get_func_name(addr)
    argnum, ret = get_argnum_and_ret(addr)
    generate_for_func(soName, functionName, addr, argnum, ret)


def generate_for_inline_by_address(addr):
    soName = idaapi.get_root_filename()
    generate_for_inline(soName, addr)


# 测试
# generate_for_func_by_address(0x3dd51)
# generate_for_inline_by_address(0x2bd80)

```

将代码完整拷贝到 IPyIDA 中运行，结果符合预期。或许我们将函数 Hook 和 inline Hook 做统一处理。如果传入地址是函数的首地址，那么就做函数 Hook，如果传入地址并非函数首地址，就做 inline Hook 。

函数首地址可以通过获取函数属性取到

```
In [109]: import idc

In [110]: idc.get_func_attr(
```

```
Signature: idc.get_func_attr(ea, attr)
Docstring:
Get a function attribute

@param ea: any address belonging to the function
@param attr: one of FUNCATTR_... constants

@return: BADADDR - error otherwise returns the attribute value
File:      d:\baidunetdiskdownload\ida_pro_v7.5_portable\ida_pro_v7.5_portable
\python\3\idc.py
Type:      function
```

函数可查询的属性很多，下列一部分

Python | 复制代码

```
FUNCATTR_START   =  0     # readonly: 函数起始地址
FUNCATTR_END     =  4     # readonly: 函数结束地址
FUNCATTR_FLAGS   =  8     # 函数的 flags
FUNCATTR_FRAME   = 16     # readonly: function frame id
FUNCATTR_FRSIZE  = 20     # readonly: 本地变量所占大小
FUNCATTR_FRREGS  = 24     # readonly: 保存寄存器区域所占大小
```

运行结果如下

IPyIDA | Python | 复制代码

```
hex(idc.get_func_attr(0x2288, 0))
Out[111]: '0x2288'

hex(idc.get_func_attr(0x110b0, 0))
Out[112]: '0x110a8'
```

调整我们的脚本代码，增加一层封装，顺便限制地址必须是属于某个，对着数据块做 Hook 显然不符合逻辑。

Python | 复制代码

```
def generate_snippet(addr):
    startAddress = idc.get_func_attr(addr, 0)
    if startAddress == addr:
        generate_for_func_by_address(addr)
    elif startAddress == idc.BADADDR:
        print("不在函数内")
    else:
        generate_for_inline_by_address(addr)
```

还有一个小问题，ARM32 架构下，THUMB 模式中 Hook 地址需要+1。首先是如何判断正在处理的文件是ARM32还是ARM64

Python | 复制代码

```
import idaapi

idaapi.get_inf_structure().is_64bit()
Out[3]: False

idaapi.get_inf_structure().is_32bit()
Out[4]: True
```

其次需要判断某条指令是ARM还是Thumb，静态分析中，可以通过T标志位判断

`idc.get_sreg(addr, "T") thumb返回1，否则返回0`

可以通过如下逻辑进行处理

Python | 复制代码

```
    if idaapi.get_inf_structure().is_64bit():
        offset = addr
    else:
        offset = addr + idc.get_sreg(addr, "T")
```

总体代码如下

Python | 复制代码

```python
from string import Template
import ida_lines
import idaapi
import idc

hook_function_template = """
function hook_$functionName(){
    var base_addr = Module.findBaseAddress("$soName");

    Interceptor.attach(base_addr.add($offset), {
        onEnter(args) {
            console.log("call $functionName");
            $args
        },
        onLeave(retval) {
            $result
            console.log("leave $functionName");
        }
    });
}
"""

inline_hook_template = """
function hook_$offset(){
    var base_addr = Module.findBaseAddress("$soName");

    Interceptor.attach(base_addr.add($offset), {
        onEnter(args) {
            console.log("call $offset");
            console.log(JSON.stringify(this.context));
        },
    });
}
"""

logTemplate = 'console.log("arg$index:"+args[$index]);\n'


def generate_printArgs(argNum):
    if argNum == 0:
        return "// no args"
    else:
        temp = Template(logTemplate)
        logText = ""
        for i in range(argNum):
            logText += temp.substitute({'index': i})
            logText += "            "
        return logText


def generate_for_func(soName, functionName, address, argNum, hasReturn):
    # 根据参数个数打印
    argsPrint = generate_printArgs(argNum)
    # 根据是否有返回值判断是否打印retval
    retPrint = "// no return"
    if hasReturn:
        retPrint = "console.log(retval);"
    # 使用Python提供的Template字符串模板方法
    temp = Template(hook_function_template)
    offset = getOffset(address)
    result = temp.substitute(
        {'soName': soName, "functionName": functionName, "offset": hex(offset), "args": argsPrint, "result": retPrint})
    print(result)


def getOffset(address):
    if idaapi.get_inf_structure().is_64bit():
        return address
    else:
        return address + idc.get_sreg(address, "T")


def generate_for_inline(soName, address):
```

```
    offset = getOffset(address)
    temp = Template(inline_hook_template)
    result = temp.substitute({'soName': soName, "offset": hex(offset)})
    print(result)


def get_argnum_and_ret(address):
    cfun = idaapi.decompile(address)
    argnum = len(cfun.arguments)
    ret = True
    dcl = ida_lines.tag_remove(cfun.print_dcl())
    if (dcl.startswith("void ") is True) & (dcl.startswith("void *") is False):
        ret = False
    return argnum, ret


def generate_for_func_by_address(addr):
    soName = idaapi.get_root_filename()
    functionName = idaapi.get_func_name(addr)
    argnum, ret = get_argnum_and_ret(addr)
    generate_for_func(soName, functionName, addr, argnum, ret)


def generate_for_inline_by_address(addr):
    soName = idaapi.get_root_filename()
    generate_for_inline(soName, addr)


def generate_snippet(addr):
    startAddress = idc.get_func_attr(addr, 0)
    if startAddress == addr:
        generate_for_func_by_address(addr)
    elif startAddress == idc.BADADDR:
        print("不在函数内")
    else:
        generate_for_inline_by_address(addr)


# TEST
generate_snippet(0x3dd50)
generate_snippet(0x2bd80)
```

应该说基本功能已经实现了，那么我们会在逆向分析中使用这个辅助脚本吗？我想并不会，它在使用上存在两个麻烦的地方。首先，当需要使用时，我们需要找到这个脚本，运行在 IDA 的 Python 环境里。其次，需要根据具体地址调用 generate_snippet 函数。

先看第二个问题是否存在优化空间，比如，我们是否能在反汇编代码界面双击，就根据当前所处的汇编地址生成 generate_snippet 调用?

这涉及到 IDA 中 Hook 的问题，这里所说的 Hook 不是 Hook 我们分析的二进制文件，而是 Hook IDA 中各种事件和时机。IDA 中提供了对自身充分的 Hook ，让我们可以拦截和观测它自身在UI界面、反汇编过程、反编译过程、调试等过程中的每一点风吹草动，包括 UI_HOOKS、DBG_HOOKS、Hexrays_HOOKS、VIEW_HOOKS、DBG_HOOKS、IDP_HOOKS 等等。

本篇主要讨论 View_Hooks，它意味着对UI事件的监听。原始的 View_Hooks 类是下面这样

Python | 复制代码

```
class View_Hooks:

    def view_activated(self, *args):
        pass

    def view_deactivated(self, *args):
        pass

    def view_keydown(self, *args):
        pass

    def view_click(self, *args):
        pass

    def view_dblclick(self, *args):
        pass

    # ... 省略
```

其中什么功能都没有实现，我们想实现一些自定义的操作，比如当鼠标双击时记录当前地址，可以继承这个类并定制我们需要的功能。让我们看一下 View_Hooks 给我们提供了哪些具体的 HOOK？如何继承和定义自己的 View_Hooks？

view_activated / view_deactivated： 界面打开和离开界面

读者可以运行下面的代码， 并尝试在伪代码界面、十六进制界面、汇编界面来回切换。当一个界面被展示在最前端时是 activated 状态，被其他界面覆盖不可见时是 deactivated 状态。如果刷新界面，其实是一个短暂的从可见到不可见再到可见的状态，即 deactivated + activated 。

Python | 复制代码

```python
import idaapi


class Hook(idaapi.View_Hooks):
    def view_activated(self, view):
        viewName = idaapi.get_widget_title(view)
        print("view active:"+viewName)

    def view_deactivated(self, view):
        viewName = idaapi.get_widget_title(view)
        print("view deactive:"+viewName)


myViewHook = Hook()
# 开启自定义的View Hook
myViewHook.hook()

```

view_created 和 view_close 关注的不是界面的可见和不可见，而是界面的创建与关闭。

Python | 复制代码

```python
def view_created(self, view):
    viewName = idaapi.get_widget_title(view)
    print("view create:"+viewName)

def view_close(self, view):
    viewName = idaapi.get_widget_title(view)
    print("view close:"+viewName)
```

读者可以关闭伪代码界面、十六进制界面、汇编界面这些主要的界面，然后再通过View-open-subviews子菜单重新打开这些界面，来观察这一过程。

view_keydown 则会监控当鼠标悬停在IDA交互界面上时键盘的输入，view_keydown参数中包含了具体输入的值

Python | 复制代码

```python
def view_keydown(self, view ,key ,state):
    print("keydown:"+chr(key))
```

需要注意，并不是所有的按键输入都会被我们打印出来，因为一些按键作为默认或插件的快捷键，优先级更高，会走入它们的逻辑。

view_click 会监控每一次单击，view_dblclick 会监控连续的两次单击，即双击。

Python | 复制代码

```python
def view_click(self, view, event):
    print("one click")

def view_dblclick(self, view ,event):
    print("double click")
```

如果要做某种自定义的逻辑，用双击比较好，单击太频繁了不容易判断。需要注意的是，单击触发的时机是鼠标松开的时机，而非按压的时机，这在后面会有用。

view_curpos 监控鼠标位置改变，侧重于观测鼠标移动给界面带来的变动。

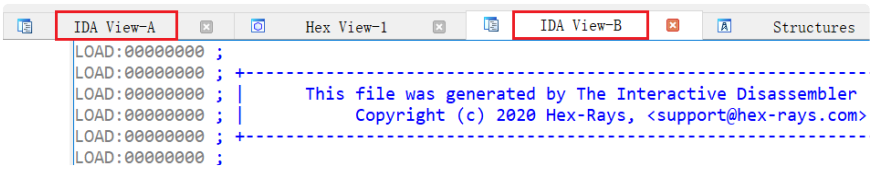
view_mouse_moved 单纯监控鼠标的移动操作，因此移动一次可能会触发几十次 view_mouse_moved 。

Python | 复制代码

```python
def view_curpos(self, view):
    print("curpos changed")

def view_mouse_moved(self, view ,event):
    print("moved mouse")

```

以上是 View_Hooks 提供的一些主要的界面Hook。

我们如何使用它呢？当发生一次鼠标双击操作时，就打印光标所处位置的地址。为了防止误判，还应该判断当前界面是否为反汇编界面。在前面的代码测试中，有介绍过如何获取当前界面的名字： `viewName = idaapi.get_widget_title(view)` viewName 其实就是界面顶部所呈现的名字。



反汇编界面对应于—— IDA View-XXX。我们可以对 viewName 字符串做正则匹配，进而判断当前界面是否是反汇编界面，诸如 `viewName.contains("IDA View")` 。但这里有个更优雅一些的写法，直接通过类型判断，代码如下

Python | 复制代码

```python
def view_dblclick(self, view ,event):
    widgetType = idaapi.get_widget_type(view)
    if widgetType == idaapi.BWN_DISASM:
        # 后续逻辑
        pass
```

下面是如何获取光标所处地址，代码非常简单 `address = idaapi.get_screen_ea()` 整体调整如下

Python | 复制代码

```python
from string import Template
import ida_lines
import idaapi
import idc

hook_function_template = """
function hook_$functionName(){
    var base_addr = Module.findBaseAddress("$soName");

    Interceptor.attach(base_addr.add($offset), {
        onEnter(args) {
            console.log("call $functionName");
            $args
        },
        onLeave(retval) {
            $result
            console.log("leave $functionName");
        }
    });
}
"""

inline_hook_template = """
function hook_$offset(){
    var base_addr = Module.findBaseAddress("$soName");

    Interceptor.attach(base_addr.add($offset), {
        onEnter(args) {
            console.log("call $offset");
            console.log(JSON.stringify(this.context));
        },
    });
}
"""

logTemplate = 'console.log("arg$index:"+args[$index]);\n'


def generate_printArgs(argNum):
    if argNum == 0:
        return "// no args"
    else:
        temp = Template(logTemplate)
        logText = ""
        for i in range(argNum):
            logText += temp.substitute({'index': i})
            logText += "            "
        return logText


def generate_for_func(soName, functionName, address, argNum, hasReturn):
    # 根据参数个数打印
    argsPrint = generate_printArgs(argNum)
    # 根据是否有返回值判断是否打印retval
    retPrint = "// no return"
    if hasReturn:
        retPrint = "console.log(retval);"
    # 使用Python提供的Template字符串模板方法
    temp = Template(hook_function_template)
    offset = getOffset(address)
    result = temp.substitute(
        {'soName': soName, "functionName": functionName, "offset": hex(offset), "args": argsPrint, "result": retPrint})
    print(result)


def getOffset(address):
    if idaapi.get_inf_structure().is_64bit():
        return address
    else:
        return address + idc.get_sreg(address, "T")


def generate_for_inline(soName, address):
```

```
74      offset = getOffset(address)
75      temp = Template(inline_hook_template)
76      result = temp.substitute({'soName': soName, "offset": hex(offset)})
77      print(result)
78
79
80  def get_argnum_and_ret(address):
81      cfun = idaapi.decompile(address)
82      argnum = len(cfun.arguments)
83      ret = True
84      dcl = ida_lines.tag_remove(cfun.print_dcl())
85      if (dcl.startswith("void ") is True) & (dcl.startswith("void *") is False):
86          ret = False
87      return argnum, ret
88
89
90  def generate_for_func_by_address(addr):
91      soName = idaapi.get_root_filename()
92      functionName = idaapi.get_func_name(addr)
93      argnum, ret = get_argnum_and_ret(addr)
94      generate_for_func(soName, functionName, addr, argnum, ret)
95
96
97  def generate_for_inline_by_address(addr):
98      soName = idaapi.get_root_filename()
99      generate_for_inline(soName, addr)
100
101
102 def generate_snippet(addr):
103     startAddress = idc.get_func_attr(addr, 0)
104     if startAddress == addr:
105         generate_for_func_by_address(addr)
106     elif startAddress == idc.BADADDR:
107         print("不在函数内")
108     else:
109         generate_for_inline_by_address(addr)
110
111
112 class Hook(idaapi.View_Hooks):
113     def view_dblclick(self, view ,event):
114         widgetType = idaapi.get_widget_type(view)
115         if widgetType == idaapi.BWN_DISASM:
116             address = idaapi.get_screen_ea()
117             generate_snippet(address)
118
119
120 myViewHook = Hook()
121 # 开启自定义的View Hook
122 myViewHook.hook()
123
```

似乎还不错，现在还需要解决上面提的另一个问题，即当需要使用时，我们需要找到这个脚本，然后运行在 IDA 的 Python 环境里，这很麻烦。为了避免这样的情况，我们需要将这个脚本转成插件。

在通过 IDAPython 编写 IDA 插件时，我们需要定义一个名为 PLUGIN_ENTRY 的函数，该函数必须返回继承了 plugin_t 插件类的实例。正因如此，我们一般把插件类命名为xxx_Plugin_t。

Python | 复制代码

```
1  class GenFrida_Plugin_t(idaapi.plugin_t):
2      pass
3
4  # register IDA plugin
5  def PLUGIN_ENTRY():
6      return XXX_Plugin_t()
```

接下来我们需要初始化插件相关的一系列成员变量，以及实现插件类的三个基本方法。

Python | 复制代码

```python
import idaapi
import idc
from ida_idaapi import plugin_t

class GenFrida_Plugin_t(plugin_t):
    # 关于插件的注释
    # 当鼠标浮于菜单插件上方时，IDA左下角所示
    comment = "A Toy Plugin for Generating Frida Code"
    # 帮助信息，我们选择不填
    help = "unknown"
    # 插件的特性，是一直在内存里，还是运行一下就退出，等等
    flags = idaapi.PLUGIN_KEEP
    # 插件的名字
    wanted_name = "ShowFridaCode"
    # 快捷键，我们选择置空不弄
    wanted_hotkey = ""

    # 插件刚被加载到IDA内存中
    # 这里适合做插件的初始化工作
    def init(self):
        print("ShowFridaCode init")
        return idaapi.PLUGIN_KEEP

    # 插件运行中
    # 这里是主要逻辑
    def run(self, arg):
        print("ShowFridaCode run")

    # 插件卸载退出的时机
    # 这里适合做资源释放
    def term(self):
        pass


# register IDA plugin
def PLUGIN_ENTRY():
    return GenFrida_Plugin_t()
```

如果你想做一个非常精良的插件，那么最好仔细考虑每一个成员变量和函数如何设置，使得用户体验与效率最优，但如果只是《能用就行》，那么上面注释所述就足够了。插件的安装也很简单，将这个 python 文件拷贝到对应IDA 的 plugins 目录下即可。值得一提的是，脚本可以即写即用，而插件需要重新打开IDA才会加载与生效。

现在的完整代码如下

Python | 复制代码

```python
from string import Template
import ida_lines
import idaapi
import idc
from ida_idaapi import plugin_t

hook_function_template = """
function hook_$functionName(){
    var base_addr = Module.findBaseAddress("$soName");

    Interceptor.attach(base_addr.add($offset), {
        onEnter(args) {
            console.log("call $functionName");
            $args
        },
        onLeave(retval) {
            $result
            console.log("leave $functionName");
        }
    });
}
"""

inline_hook_template = """
function hook_$offset(){
    var base_addr = Module.findBaseAddress("$soName");

    Interceptor.attach(base_addr.add($offset), {
        onEnter(args) {
            console.log("call $offset");
            console.log(JSON.stringify(this.context));
        },
    });
}
"""

logTemplate = 'console.log("arg$index:"+args[$index]);\n'


def generate_printArgs(argNum):
    if argNum == 0:
        return "// no args"
    else:
        temp = Template(logTemplate)
        logText = ""
        for i in range(argNum):
            logText += temp.substitute({'index': i})
            logText += "            "
        return logText


def generate_for_func(soName, functionName, address, argNum, hasReturn):
    # 根据参数个数打印
    argsPrint = generate_printArgs(argNum)
    # 根据是否有返回值判断是否打印retval
    retPrint = "// no return"
    if hasReturn:
        retPrint = "console.log(retval);"
    # 使用Python提供的Template字符串模板方法
    temp = Template(hook_function_template)
    offset = getOffset(address)
    result = temp.substitute(
        {'soName': soName, "functionName": functionName, "offset": hex(offset), "args": argsPrint, "result": retPrint})
    print(result)


def getOffset(address):
    if idaapi.get_inf_structure().is_64bit():
        return address
    else:
        return address + idc.get_sreg(address, "T")


```

```
74  def generate_for_inline(soName, address):
75      offset = getOffset(address)
76      temp = Template(inline_hook_template)
77      result = temp.substitute({'soName': soName, "offset": hex(offset)})
78      print(result)
79
80
81  def get_argnum_and_ret(address):
82      cfun = idaapi.decompile(address)
83      argnum = len(cfun.arguments)
84      ret = True
85      dcl = ida_lines.tag_remove(cfun.print_dcl())
86      if (dcl.startswith("void ") is True) & (dcl.startswith("void *") is False):
87          ret = False
88      return argnum, ret
89
90
91  def generate_for_func_by_address(addr):
92      soName = idaapi.get_root_filename()
93      functionName = idaapi.get_func_name(addr)
94      argnum, ret = get_argnum_and_ret(addr)
95      generate_for_func(soName, functionName, addr, argnum, ret)
96
97
98  def generate_for_inline_by_address(addr):
99      soName = idaapi.get_root_filename()
100     generate_for_inline(soName, addr)
101
102
103 def generate_snippet(addr):
104     startAddress = idc.get_func_attr(addr, 0)
105     if startAddress == addr:
106         generate_for_func_by_address(addr)
107     elif startAddress == idc.BADADDR:
108         print("不在函数内")
109     else:
110         generate_for_inline_by_address(addr)
111
112
113 class Hook(idaapi.View_Hooks):
114     def view_dblclick(self, view ,event):
115         widgetType = idaapi.get_widget_type(view)
116         if widgetType == idaapi.BWN_DISASM:
117             address = idaapi.get_screen_ea()
118             generate_snippet(address)
119
120
121 class GenFrida_Plugin_t(plugin_t):
122     # 关于插件的注释
123     # 当鼠标浮于菜单插件上方时，IDA左下角所示
124     comment = "A Toy Plugin for Generating Frida Code"
125     # 帮助信息，我们选择不填
126     help = "unknown"
127     # 插件的特性，是一直在内存里，还是运行一下就退出，等等
128     flags = idaapi.PLUGIN_KEEP
129     # 插件的名字
130     wanted_name = "ShowFridaCode"
131     # 快捷键，我们选择置空不弄
132     wanted_hotkey = ""
133
134     # 插件刚被加载到IDA内存中
135     # 这里适合做插件的初始化工作
136     def init(self):
137         print("ShowFridaCode init")
138         return idaapi.PLUGIN_KEEP
139
140     # 插件运行中
141     # 这里是主要逻辑
142     def run(self, arg):
143         print("ShowFridaCode run")
144         global myViewHook
145         myViewHook = Hook()
146         myViewHook.hook()
147
148     # 插件卸载退出的时机
```

```
    # 这里适合做资源释放
    def term(self):
        pass


# register IDA plugin
def PLUGIN_ENTRY():
    return GenFrida_Plugin_t()

```

就像使用其他插件一样，IDA Edit - plugins 菜单栏中找到我们的 showfridacode 插件并点击，在当前环境中插件就顺利激活了。

在Hook Native时，我们常常通过Spawn抢占两个较早的时机，1是 JNIOnLoad 前 2是 init_proc 以及 init_array 前。我们可以给这个小插件添加两个新模板，当插件在当前打开的项目中被首次触发使用时，打印这两个时机对应的模板。

Python | 复制代码

```
dlopenAfter_template = """
var android_dlopen_ext = Module.findExportByName(null, "android_dlopen_ext");
if(android_dlopen_ext != null){
    Interceptor.attach(android_dlopen_ext,{
        onEnter: function(args){
            var soName = args[0].readCString();
            if(soName.indexOf("$soName") !== -1){
                this.hook = true;
            }
        },
        onLeave: function(retval){
            if(this.hook) {
                this.hook = false;
                dlopentodo();
            }
        }
    });
}

function dlopentodo(){
    //todo
}
"""

init_template = """
function hookInit(){
    var linkername;
    var alreadyHook = false;
    var call_constructor_addr = null;
    var arch = Process.arch;
    if (arch.endsWith("arm")) {
        linkername = "linker";
    } else {
        linkername = "linker64";
    }

    var symbols = Module.enumerateSymbolsSync(linkername);
    for (var i = 0; i < symbols.length; i++) {
        var symbol = symbols[i];
        if (symbol.name.indexOf("call_constructor") !== -1) {
            call_constructor_addr = symbol.address;
        }
    }

    if (call_constructor_addr.compare(NULL) > 0) {
        console.log("get construct address");
        Interceptor.attach(call_constructor_addr, {
            onEnter: function (args) {
                if(alreadyHook === false){
                    const targetModule = Process.findModuleByName("$soName");
                    if (targetModule !== null) {
                        alreadyHook = true;
                        inittodo();
                    }
                }
            }
        });
    }
}

function inittodo(){
    //todo
}
"""
```

完整代码如下所示

Python | 复制代码

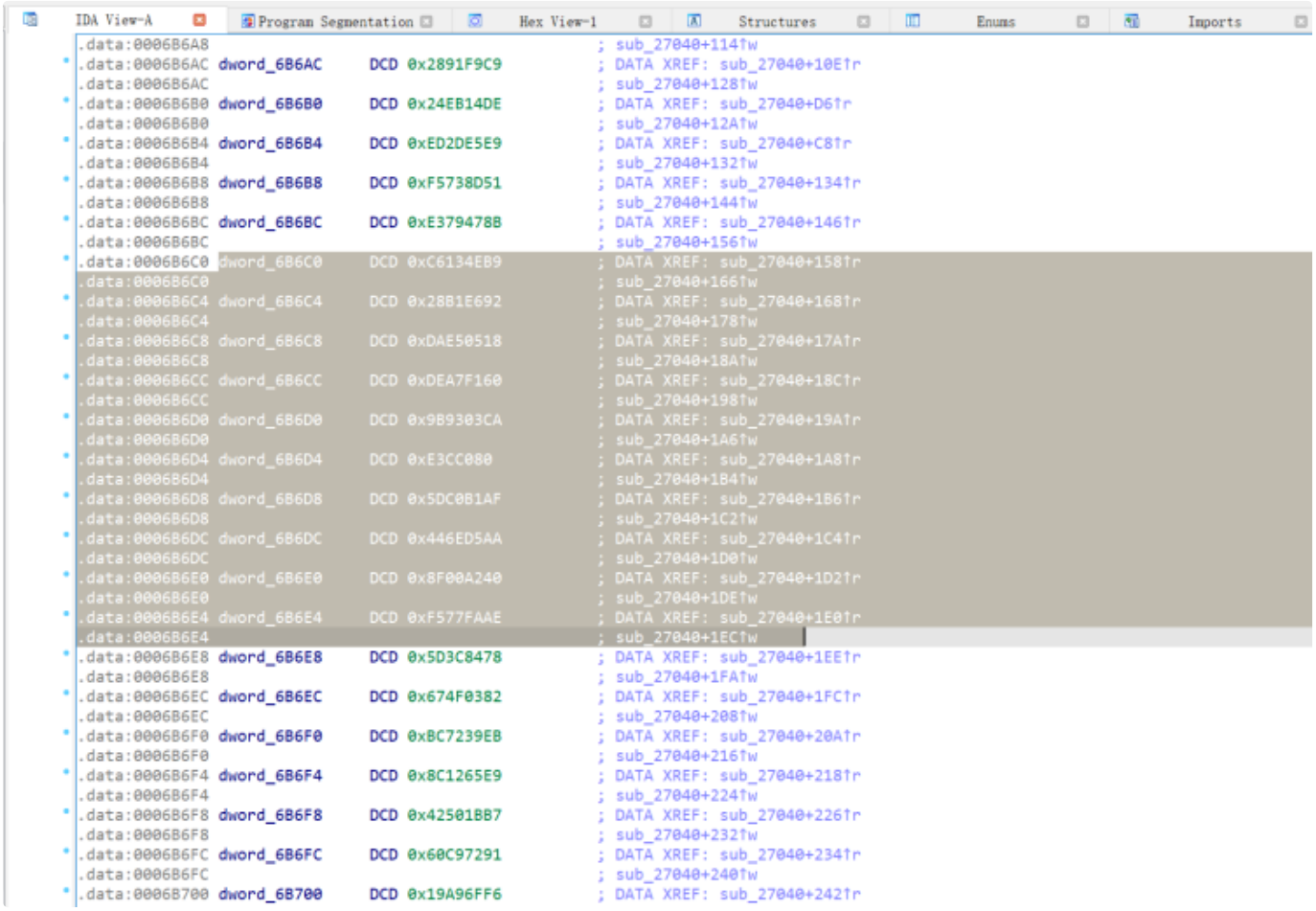
```
dlopenAfter_template =
var android_dlopen_ext = Module.findExportByName(null, "android_dlopen_ext");
if(android_dlopen_ext != null){
    Interceptor.attach(android_dlopen_ext,{
        onEnter: function(args){
            var soName = args[0].readCString();
            if(soName.indexOf("$soName") !== -1){
                this.hook = true;
            }
        },
        onLeave: function(retval){
            if(this.hook) {
                this.hook = false;
                dlopentodo();
            }
        }
    });
}

function dlopentodo(){
    //todo
}
"""

init_template = """
function hookInit(){
    var linkername;
    var alreadyHook = false;
    var call_constructor_addr = null;
    var arch = Process.arch;
    if (arch.endsWith("arm")) {
        linkername = "linker";
    } else {
        linkername = "linker64";
    }
```

```
        return logText


def generate_for_func(soName, functionName, address, argNum, hasReturn):
    # 根据参数个数打印
    argsPrint = generate_printArgs(argNum)
    # 根据是否有返回值判断是否打印retval
    retPrint = "// no return"
    if hasReturn:
        retPrint = "console.log(retval);"
    # 使用Python提供的Template字符串模板方法
    temp = Template(hook_function_template)
    offset = getOffset(address)
    result = temp.substitute(
        {'soName': soName, "functionName": functionName, "offset": hex(offset), "args": argsPrint, "result": retPrint})
    print(result)


def getOffset(address):
    if idaapi.get_inf_structure().is_64bit():
        return address
    else:
        return address + idc.get_sreg(address, "T")


def generate_for_inline(soName, address):
    offset = getOffset(address)
    temp = Template(inline_hook_template)
    result = temp.substitute({'soName': soName, "offset": hex(offset)})
    print(result)


def get_argnum_and_ret(address):
    cfun = idaapi.decompile(address)
    argnum = len(cfun.arguments)
    ret = True
```

```
            if not initialized:
                initialized = True
                generateInitCode()
            address = idaapi.get_screen_ea()
            generate_snippet(address)


class GenFrida_Plugin_t(plugin_t):
    # 关于插件的注释
    # 当鼠标浮于菜单插件上方时，IDA左下角所示
    comment = "A Toy Plugin for Generating Frida Code"
    # 帮助信息，我们选择不填
    help = "unknown"
    # 插件的特性，是一直在内存里，还是运行一下就退出，等等
    flags = idaapi.PLUGIN_KEEP
    # 插件的名字
    wanted_name = "ShowFridaCode"
    # 快捷键，我们选择置空不弄
    wanted_hotkey = ""

    # 插件刚被加载到IDA内存中
    # 这里适合做插件的初始化工作
    def init(self):
        print("ShowFridaCode init")
        return idaapi.PLUGIN_KEEP

    # 插件运行中
    # 这里是主要逻辑
    def run(self, arg):
        print("ShowFridaCode run")
        global myViewHook
        myViewHook = Hook()
        myViewHook.hook()

    # 插件卸载退出的时机
    # 这里适合做资源释放
    def term(self):
        pass


initialized = False
# register IDA plugin
def PLUGIN_ENTRY():
    return GenFrida_Plugin_t()
```

应该说，目前已经基本可用了。但还有两个点可以做优化。

1是如果用户在反汇编界面框选出一块数据，那么我们可以生成对应的 Frida dump，这在字符串解密一类的场景可能有用。



模板像下面这样

JavaScript | 复制代码

```javascript
function dumpAddr() {
    var base_addr = Module.findBaseAddress("$soName");
    var dump_addr = base_addr.add($addr);
    console.log(dump_addr, {length: $length});
}
```

框选这个过程本身就是一次单击操作，可以拦截到这个时机，再通过 API 获得其范围

Python | 复制代码

```python
hex(idc.read_selection_start())
Out[22]: '0xff638'

hex(idc.read_selection_start())
Out[23]: '0xffffffffffffffff'
```

除此之外也可以使用 `idaapi.read_range_selection` API 获得包含起始和结束范围的列表。增加上述逻辑后，新代码如下所示

Python | 复制代码

```
init_template = `
function hookInit(){
    var linkername;
    var alreadyHook = false;
    var call_constructor_addr = null;
    var arch = Process.arch;
    if (arch.endsWith("arm")) {
        linkername = "linker";
    } else {
        linkername = "linker64";
    }
```

```
137             {'soName': soName, "functionName": functionName, "offset": hex(offset), "args": argsPrint, "result": retPrint})
138         print(result)
139 
140 
141 def getOffset(address):
142     if idaapi.get_inf_structure().is_64bit():
143         return address
144     else:
145         return address + idc.get_sreg(address, "T")
146 
147 
148 def generate_for_inline(soName, address):
```

```
            start = idc.read_selection_start()
            end = idc.read_selection_end()
            if (start != idaapi.BADADDR) and (end != idaapi.BADADDR):
                length = end - start
                generate_dump_script(start, length)


class GenFrida_Plugin_t(plugin_t):
    # 关于插件的注释
    # 当鼠标浮于菜单插件上方时，IDA左下角所示
    comment = "A Toy Plugin for Generating Frida Code"
    # 帮助信息，我们选择不填
    help = "unknown"
    # 插件的特性，是一直在内存里，还是运行一下就退出，等等
    flags = idaapi.PLUGIN_KEEP
    # 插件的名字
    wanted_name = "ShowFridaCode"
    # 快捷键，我们选择置空不弄
    wanted_hotkey = ""

    # 插件刚被加载到IDA内存中
    # 这里适合做插件的初始化工作
    def init(self):
        print("ShowFridaCode init")
        return idaapi.PLUGIN_KEEP

    # 插件运行中
    # 这里是主要逻辑
    def run(self, arg):
        print("ShowFridaCode run")
        global myViewHook
        myViewHook = Hook()
        myViewHook.hook()

    # 插件卸载退出的时机
    # 这里适合做资源释放
    def term(self):
        pass


initialized = False
# register IDA plugin
def PLUGIN_ENTRY():
    return GenFrida_Plugin_t()

```

2是对inline hook的处理，可以更细腻一些，比如下面这个汇编片段

Python | 复制代码

```
.text:00007DAA                SUB             SP, SP, #0x18
.text:00007DAC                MOVS            R0, #0x10
.text:00007DAE                MOVS            R1, #3
.text:00007DB0                LDR.W           R2, [R7,#var_34]
.text:00007DB4                MOVS            R3, #0x80
.text:00007DB6                STRD.W          R2, R1, [SP,#0x68+var_68]
.text:00007DBA                MOV             R1, R11
.text:00007DBC                STRD.W          R0, R9, [SP,#0x68+var_60]
.text:00007DC0                MOV             R0, R10
.text:00007DC2                STR             R6, [SP,#0x68+var_58]
.text:00007DC4                LDR             R2, =(byte_69650 - 0x7DCA)
.text:00007DC6                ADD             R2, PC  ; byte_69650
.text:00007DC8                BL              sub_43764
```

如果用户在 0x7DC2 处 inline Hook，我们是否可以揣测出用户的意图？这是很难的，他可能想查看 R0、R1 或者其他任意寄存器，因此我们只能宽泛的打印完整寄存器环境。

`console.log(JSON.stringify(this.context));`

但如果用户在 0x7DC8 处 inline Hook，我们是否可以揣测出用户的意图呢？其实是可以的，在大多数情况下，hook 子函数调用位置处的代码，是为了查看这个子函数调用的入参。为什么不直接 Hook sub_43764 函数？因为 sub_43764 可能在非常多的位置发生了调用，但我们此时只想关注当前地址发生的调用，那么就会采用这样的inline hook 避免一些干扰输出，那么宽泛的打印完整寄存器环境并非一种必要，我们可以直接打印所调用函数。

简而言之，当 inline hook 一个子函数调用的时机时，我们可以像hook 函数那样直接打印参数，而不是宽泛的打印寄存器信息。

那么首先第一个问题就是，我们如何判断某条指令发起了子程序调用？有很多种代码模式可以发起子程序调用，只需要保证 PC 跳转 + LR 正确保存。但一般情况下，编译器并不会这么做，而是采用固定的指令发起子程序调用。在 X86 中，这指令一般是 call ，在ARM中，这指令是 BL 或 BLX ，在 ARM64 中，这指令是 BL 或 BLR 。

我们主要适配ARM和ARM64，那么可以解析当前指令是否是对应的指令，来判断当前是否发起子程序调用。但IDA提供了更优雅的办法，可以直接调用它的API来判断。

Python | 复制代码

```
# Is the instruction a "call"?
idaapi.is_call_insn(ea)
```

那么第二个问题就是如何解析call指向的地址，需要注意，这里还需要考虑是否能直接获取到真实地址

Python | 复制代码

```
.text:000053F6 25 F0 4F FC              BL              sub_2AC98
.text:000053FA 02 B0                    ADD             SP, SP, #8
.text:000053FC 01 46                    MOV             R1, R0
.text:000053FE 48 46                    MOV             R0, R9
.text:00005400 C0 47                    BLX             R8
```

比如上面代码片段中的 0x5400 处，在静态分析中就很难获取到真实的函数地址，但我们希望 0x53F6 处这种可以获取到地址并进一步分析。我们需要获取对应的call指令后跟着的操作数，判断其类型是立即数还是寄存器，如果是立即数，再打印其值作为子程序地址。如果是寄存器，则不做处理。

Python | 复制代码

```
hex(idc.get_operand_value(0x5400, 0)) == idc.o_imm
Out[18]: False

hex(idc.get_operand_value(0x53f6, 0))
Out[15]: '0x2ac98'
```

但实践中发现不是效果特别好，因为操作数包含许多种类型，需要分别处理。使用反汇编界面解析出的结果会更方便。

Python | 复制代码

```
import idaapi

idc.print_operand(0x9eee, 0)
Out[19]: 'sub_3DE7C'

hex(idaapi.get_name_ea(0, 'sub_3DE7C'))
Out[20]: '0x3de7c'

idc.print_operand(0xa370, 0)
Out[21]: '__aeabi_memcpy'

hex(idaapi.get_name_ea(0, '__aeabi_memcpy'))
Out[22]: '0x2048'

hex(idaapi.get_name_ea(0, 'r2'))
Out[23]: '0xffffffff'

hex(idaapi.get_name_ea(0, 'abc'))
Out[24]: '0xffffffff'
```

即解析 call 的函数其地址，如果指向寄存器或找不到对应符号，32 位二进制文件返回 0xffffffff， 64 位二进制文件返回 0xffffffffffffffff ，这是 IDA 中 BADADDR 的定义值，不用我们额外做匹配。

最终代码如下

Python | 复制代码

```python
from string import Template
import ida_lines
import idaapi
import idc
from ida_idaapi import plugin_t

hook_function_template = """
function hook_$functionName(){
    var base_addr = Module.findBaseAddress("$soName");

    Interceptor.attach(base_addr.add($offset), {
        onEnter(args) {
            console.log("call $functionName");
            $args
        },
        onLeave(retval) {
            $result
            console.log("leave $functionName");
        }
    });
}
"""

inline_hook_template = """
function hook_$offset(){
    var base_addr = Module.findBaseAddress("$soName");

    Interceptor.attach(base_addr.add($offset), {
        onEnter(args) {
            console.log("call $offset");
            console.log(JSON.stringify(this.context));
        },
    });
}
"""

logTemplate = 'console.log("arg$index:"+args[$index]);\n'

dlopenAfter_template = """
var android_dlopen_ext = Module.findExportByName(null, "android_dlopen_ext");
if(android_dlopen_ext != null){
    Interceptor.attach(android_dlopen_ext,{
        onEnter: function(args){
            var soName = args[0].readCString();
            if(soName.indexOf("$soName") !== -1){
                this.hook = true;
            }
        },
        onLeave: function(retval){
            if(this.hook) {
                this.hook = false;
                dlopentodo();
            }
        }
    });
}

function dlopentodo(){
    //todo
}
"""

init_template = """
function hookInit(){
    var linkername;
    var alreadyHook = false;
    var call_constructor_addr = null;
    var arch = Process.arch;
    if (arch.endsWith("arm")) {
        linkername = "linker";
    } else {
        linkername = "linker64";
    }
```

```
    var symbols = Module.enumerateSymbolsSync(linkername);
    for (var i = 0; i < symbols.length; i++) {
        var symbol = symbols[i];
        if (symbol.name.indexOf("call_constructor") !== -1) {
            call_constructor_addr = symbol.address;
        }
    }

    if (call_constructor_addr.compare(NULL) > 0) {
        console.log("get construct address");
        Interceptor.attach(call_constructor_addr, {
            onEnter: function (args) {
                if(alreadyHook === false){
                    const targetModule = Process.findModuleByName("$soName");
                    if (targetModule !== null) {
                        alreadyHook = true;
                        inittodo();
                    }
                }
            }
        });
    }
}

function inittodo(){
    //todo
}
"""

dump_template = """
// 由ShowFridaCode生成的dump memory
function dump_$offset() {
    var base_addr = Module.findBaseAddress("$soName");
    var dump_addr = base_addr.add($offset);
    console.log(hexdump(dump_addr, {length: $length}));
}
"""


def generate_printArgs(argNum):
    if argNum == 0:
        return "// no args"
    else:
        temp = Template(logTemplate)
        logText = ""
        for i in range(argNum):
            logText += temp.substitute({'index': i})
            logText += "            "
        return logText


def generate_for_func(soName, functionName, address, argNum, hasReturn):
    # 根据参数个数打印
    argsPrint = generate_printArgs(argNum)
    # 根据是否有返回值判断是否打印retval
    retPrint = "// no return"
    if hasReturn:
        retPrint = "console.log(retval);"
    # 使用Python提供的Template字符串模板方法
    temp = Template(hook_function_template)
    offset = getOffset(address)
    result = temp.substitute(
        {'soName': soName, "functionName": functionName, "offset": hex(offset), "args": argsPrint, "result": retPrint})
    print(result)


def getOffset(address):
    if idaapi.get_inf_structure().is_64bit():
        return address
    else:
        return address + idc.get_sreg(address, "T")


def generate_for_inline(soName, address):
```

```
    offset = getOffset(address)
    temp = Template(inline_hook_template)
    result = temp.substitute({'soName': soName, "offset": hex(offset)})
    if idaapi.is_call_insn(address):
        callAddr = idaapi.get_name_ea(0, idc.print_operand(address, 0))
        if callAddr != idaapi.BADADDR:
            callAddress = idc.get_operand_value(address, 0)
            argnum, _ = get_argnum_and_ret(callAddress)
            argsPrint = generate_printArgs(argnum)
            print(result.replace("console.log(JSON.stringify(this.context));", argsPrint))
        else:
            print(result)
    else:
        print(result)


def get_argnum_and_ret(address):
    cfun = idaapi.decompile(address)
    argnum = len(cfun.arguments)
    ret = True
    dcl = ida_lines.tag_remove(cfun.print_dcl())
    if (dcl.startswith("void ") is True) & (dcl.startswith("void *") is False):
        ret = False
    return argnum, ret


def generate_for_func_by_address(addr):
    soName = idaapi.get_root_filename()
    functionName = idaapi.get_func_name(addr)
    argnum, ret = get_argnum_and_ret(addr)
    generate_for_func(soName, functionName, addr, argnum, ret)


def generate_for_inline_by_address(addr):
    soName = idaapi.get_root_filename()
    generate_for_inline(soName, addr)


def generate_snippet(addr):
    startAddress = idc.get_func_attr(addr, 0)
    if startAddress == addr:
        generate_for_func_by_address(addr)
    elif startAddress == idc.BADADDR:
        print("不在函数内")
    else:
```

它绝对算不上一个复杂或巧妙的脚本，但确确实实能帮助我们做一些事。

三、尾声

我们一起动手写了一个小插件，这个感觉还不错叭 O(∩_∩)O。