IDA-minsc在Hex-Rays插件大赛中获得第二名（2）

Pinging / 2018-10-02 12:52:18 / 浏览数 3352 安全技术 技术讨论 顶(0) 踩(0)

■■■■ https://xz.aliyun.com/t/2841

找到RTF的token参数解析器

此时，我们仍然应该在“sub_431D38”之内。 虽然我们可以对照（Ctrl + X）指向我们在此处分配的token数组的一些指针，并最终获得处理我们的令牌及其参数的函数，但这里存在一种更简单的方法。
我们可以为数据库中的每个函数标记所有开关。 在我们处理它时，让我们计算每个开关的样例数量，这样我们就可以快速查询具有最多案例的函数。
我们可以使用function.switches■■来枚举函数中的所有开关。
这允许我们遍历函数中定义的所有开关，并返回IDA-minsc的switch_t实例，我们可以使用它来计算非默认事例的总数。
为此，我们首先定义一个函数，而该函数将标记开关的数量和函数事例总数。

```
Python>def tag_switches(ea):
Python>    count, total = 0, 0
Python>    for sw in func.switches(ea):
Python>        count += 1
Python>        total += len(sw.cases)
Python>    if count > 0:
Python>        func.tag(ea, 'switch.count', count)
Python>        func.tag(ea, 'switch.cases', total)
Python>    return
```

现在我们已经定义了一个函数。它可以标记IDA数据库中的函数，标签“switch.count”表示开关数，“switch.cases”表示案例总数，我们可以将其应用于每个函数数据库，以
让我们使用database.functions■■通过以下方法遍历所有函数：

```
Python>for ea in db.functions():
Python>    tag_switches(ea)
```

这可能需要一段时间，因此如果我们能够看到当前的进展，那将会对我们的工作更有帮助。 IDA-minsc在我们可以使用的“工具”模块中提供了一个工具。
该工具是是以callable作为参数的tools.map。 如果我们愿意使用此功能，我们可以使用以下内容：

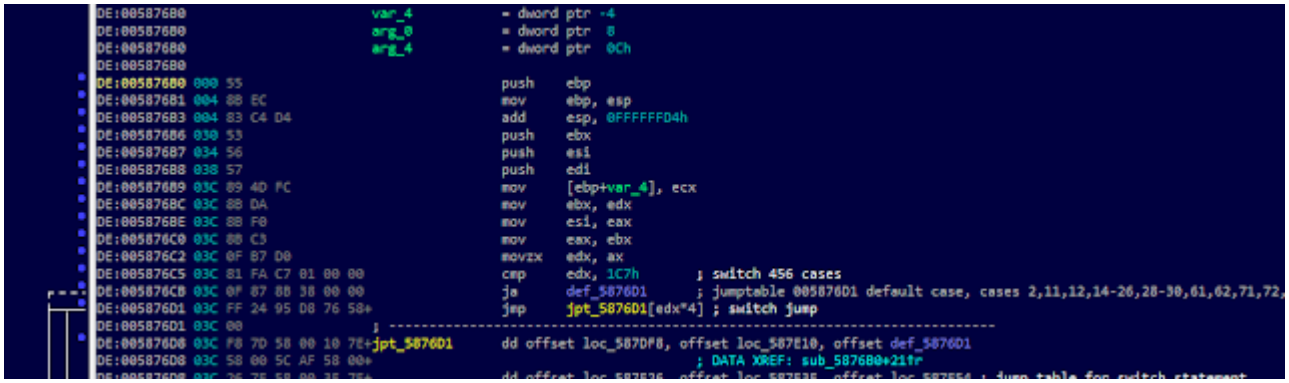
```
Python>_ = tools.map(tag_switches)
```

既然我们将每个函数都标记为开关，我们就可以查询整个数据库以便对它们进行排序。
虽然我们可以使用Python的sorted函数中的“key”关键字，但我们只是管理我们的查询结果，以便第一个条目是属于每个函数的开关总数。
之后，我们将使用sorted对它们进行排序，然后查看应该具有最多情况的元素。

```
Python>results = []
Python>for ea, tags in db.select('switch.cases'):
Python>    results.append( (tags['switch.cases'], ea) )
Python>
Python>results = sorted(results)
Python>len(results)
162
```

现在我们已经有了具有最多开关的函数的排序列表，我们可以查看最后一个元素以便找到我们所需要的内容。
让我们从最后一个结果中提取地址，然后导航到它（图6）。

```
Python>results[-1]
(294, 5797552)
Python>_, ea = results[-1]
Python>go(ea)
```



看起来我们很幸运，发现了一些看起来像解析器的东西或者真的是一个带有许多案例开关的标记器。
让我们通过调用function.tag■■来仔细检查我们的开关数。

```
Python>print func.tag('switch.count')
1
```

这里似乎只有一个开关。 让我们分析我们的开关，这样我们就可以看到它有什么样的情况。 为此，请单击其主分支的地址0x5876d1。
现在它被选中了，我们不需要将地址传递给database.get.switch■■而是让它使用当前地址。 我们将它存储到“sw”变量中。

```
Python>sw = db.get.switch()
Python>sw
<type 'switch_t{456}' at 0x5876c5> default:*0x58af5c branch[456]:*0x5876d8 register:edx
```

让我们看看这个开关中有多少个案例和默认情况。 我们将通过获取其总长度并减去其有效案例的长度来计算默认案例的数量。

```
Python>print 'number of cases:', len(sw)
number of cases: 456
Python>print 'number of default cases:', len(sw) - len(sw.cases)
number of default cases: 162
```

为了使我们更容易识别与特定案例相关联的token，我们可以使用存储在“token”列表中的内容简单地标记每个案例。
可用案例列表位于“sw”变量的“cases”属性中。 我们可以简单地调用它的.case■■方法来获取处理特定情况的程序。
我们将使用这些属性来标记每个处理程序，其中包含我们在上面分配的“token”列表中的“token”标记。

```
Python>for case in sw.cases:
Python>    handler = sw.case(case)
Python>    db.tag(handler, 'token', tokens[case])
```

然而，每种情况都可以处理多个令牌是一种问题。 这需要我们关注每个处理程序的案例。 为此，我们可以使用我们分配的“sw”变量的“handler”方法。
我们将使用“function.select”来选择我们已经标记过的所有处理程序，然后重新标记程序。
虽然function.select■■以函数地址作为参数，但由于我们要查询当前函数，因此它的地址已不在需要。

```
Python>for ea, tags in func.select('token'):
Python>    toks = []
Python>    for case in sw.handler(ea):
Python>        toks.append( tokens[case] )
Python>    db.tag(ea, 'token', toks)
```

这导致每个案例处理的token列表被标记为“token”。
如果我们需要再次找到处理程序或每个案例处理的token，我们可以再次使用function.select来获取它们。
在图7中，情况66,68,69,183和427的处理程序具有多个token值。

为了找到第一个调用指令，接下来我们要做的是遍历我们用“token”所标记过的每个地址。
我们还将寻找无条件分支。倘若我们成功找到了无条件分支，那么处理程序则被调用完成并在开关外部产生分支。
我们将使用“指令”模块去识别无条件分支或调用指令。 这个模块包含函数instruction.is_jmp和instruction.is_call。
我们将使用带有断言机制的database.address.next变量去找到与其中一个相匹配的“下一条”指令。
虽然这并不能产生100%准确的结果，但我们稍后会手动完成此操作。
因此在这种情况下准确性并不太重要。让我们使用function.select查询我们的处理程序，然后使用标记名“rtf-parameter”将处理程序的地址标记为其第一个调用指令。

```
Python>for ea, tags in func.select('token'):
Python>    next_call = db.a.next(ea, lambda ea: ins.is_jmp(ea) or ins.is_call(ea))
Python>    if ins.is_call(next_call):
Python>        db.tag(ea, 'rtf-parameter', ins.op(next_call, 0))
Python>    elif ins.is_jmp(next_call):
Python>        print "found an unconditional branch with the target: {:x}".format(ins.op(next_call, 0))
Python>    continue
```

我们在这里列出了一些对我们来说并不重要的无条件分支。 因此，让我们查询那些被标记为第一个调用指令结果。

```
Python>found = set()
Python>for ea, tags in func.select('rtf-parameter'):
Python>    found.add(tags['rtf-parameter'])
Python>
Python>found
set([4397600, 6556480, 4226632, 5797484, 5797008, 4226316, 4226640, 4397688, 5797216, 4226452, 5796288, 5797400, 5767988, 6487
```

不幸的是，这些数字没有很大用处。所以为了方便我们点击它们，让我们将它们映射为名称。

```
Python>map(func.name, found)
['sub_431A20', 'sub_640B40', 'sub_407E48', 'sub_58766C', 'sub_587490', 'sub_407D0C', 'sub_407E50', 'sub_431A78', 'sub_587560',
```

手动完成每个操作并快速查找显示的内容后，我们将结果总结如下：

```
sub_431A20 — looks like it does something with numbers and a hyphen, probably a range?
sub_640B40 — no idea what this is
sub_407E48 — fetches some property from an object and doesn't call anything
sub_58766C — looks too complex for me to care about
sub_587490 — looks too complex for me to care about
sub_407D0C — fetches some property from an array using an index
sub_407E50 — fetches some property from an array using an index
sub_431A78 — calls two functions, first one does something with spaces second one is the first in this list which does stuff w
sub_587560 — looks complex, but calls some string-related stuff
sub_407D94 — calls a function that does some allocation, probably allocating for a list
sub_5871C0 — references a few characters that look rtf specific like a "{", backslash "\" and a single quote
sub_587618 — calls a couple functions that are currently in this list
sub_580334 — calls a couple functions that are currently in this list
sub_62FC5C — looks like it resizes some object of some kind
```

回过头来看一下这些注释，sub_431A20，sub_431A78，sub_5871C0和sub_587618似乎与解析相关。 我们将这个函数列表转换回地址。
这样我们就可以将它们存储在一个集合中，看看我们交换机的哪些案例正在使用它们。 首先，我们将使用function.address■■将函数名称转换回地址。

```
Python>res = map(func.address, ['sub_431A20', 'sub_431A78', 'sub_5871c0', 'sub_587618'])
Python>match = set(res)
```

现在我们有一个匹配的集合。现在让我们收集所有的实例以及其调用的“rtf-parameter”方法。 我们还将其输出，以便我们可以单独访问它们。

```
Python>cases = []
Python>for ea, tags in func.select('rtf-parameter'):
Python>    if tags['rtf-parameter'] in match:
Python>        print hex(ea), sw.handler(ea)
Python>        for i in sw.handler(ea):
Python>            cases.append((i, func.name(tags['rtf-parameter'])))
Python>        continue
Python>    continue
Python>
Python>len(cases)
116
```

让我们从这个列表中取出一些随机样本，看看它们指向的标记，并识别出处理标记的首个函数调用。
从一些随机样本中我们发现大量的样本被称为“sub_431A20”。 我们认为这个函数似乎被用来处理一个可以包含连字符的数字。
在IDAPython命令行执行以下操作时，我们可以看到案例246,100和183调用此函数并表示以下标记。

```
Python>cases[246]
(246, 'sub_431A20')
Python>cases[100]
(100, 'sub_431A20')
Python>cases[183]
(183, 'sub_431A20')
Python>
Python>tokens[246], tokens[100], tokens[183]
```

```
('margt', 'colsx', 'highlight')
```

如果我们参考RTF文件格式的文档，我们可以发现这些标记（'\ margt'，'\ colsx'和'\ highlight'）将数字作为其参数。
我们可以肯定地假设“sub_431A20”负责处理RTF令牌之后的数字参数。
现在让我们对其进行标记，这样我们就可以查看另一个函数并确定它可能采用的参数类型。

```
Python>func.tag(0x431a20, 'synopsis', 'parses a numerical parameter belonging to an rtf token')
```

让我们将上面的“案例”列表中剩下的内容进行转储，这样我们就可以快速了解我们还没有进行研究的功能。
这其中包含token名称，以便我们可以快速引用文件格式规范以确定它可能采用的参数类型。

```
Python>for i, name in cases:
Python>    if name != 'sub_431A20':
Python>        print i, tokens[i], name
Python>    continue
27 b sub_587618
63 caps sub_587618
105 contextualspace sub_431A78
114 deleted sub_431A78
123 embo sub_587618
186 hyphauto sub_431A78
187 hyphcaps sub_431A78
190 hyphpar sub_431A78
191 i sub_587618
193 impr sub_587618
232 listsimple sub_431A78
270 outl sub_587618
346 scaps sub_587618
363 shad sub_587618
373 strike sub_587618
423 u sub_5871C0
426 ul sub_431A78
```

查看“\ b”，“\ caps”和“\ i”标记的文档后，我们发现这些标记可以使用可选的“0”作为参数用以关闭它们。
因此我们可以假设“sub_587618”是用于切换这些token参数的函数。 我们也标记这个功能。

```
Python>func.tag(0x587618, 'synopsis', 'parses an rtf parameter that can be toggled with "0"')
```

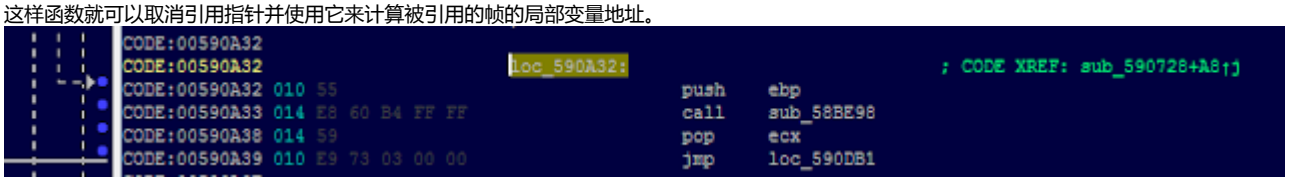
我们发出的列表中“\ hyphauto”，“\ hyphcaps”和“\ hyphpar”显现出来。 这些标记采用单个数字参数，即“1”或“0”来切换它。
这也是一个切换，但此参数是必需的。 让我们用我们新发现的知识来标记这一点。

```
Python>func.tag(0x431a78, 'synopsis','parses an rtf parameter that can be toggled with "0" or "1"')
```

现在我们已经确定了许多函数的语义。我们已经快速确定了Atlantis支持哪些token，并且可以使用此信息进行模糊特定目标。

关闭处理

Delphi 2009中引入了一种称为“匿名方法”的新功能。 此功能引入了对Delphi编程语言中闭包的支持。 闭包将捕获包含块的局部变量。
这允许闭包内的代码能够修改不同的函数的变量。
如图8所示，在Delphi生成的程序集中，■ebp寄存器中的帧指针作为参数传递给函数。
这样函数就可以取消引用指针并使用它来计算被引用的帧的局部变量地址。



这对于逆向工程师来说难度很大，因为在某些情况下局部变量通常在不同的函数中进行初始化。
要使用调试器跟踪此情况，逆向工程师可能会尝试确定变量的范围，然后使用硬件断点来标识它的第一个函数。
然而如果想要静态地去处理，这将会成为一个需要我们克服的难题。

这些局部变量的用法类似于以下代码。 在[16]中，帧从参数中提取并存储在■eax寄存器中。
这在[17]和[18]方法中重复多次以便取消引用每个调用者的帧指针的堆栈。 最后我们使用[19]，用以获取所确定的帧的局部变量。
在Atlantis应用中，这种类型的构造是非常常见的。但其也可能难以被用户管理。

```
CODE:0058BED8 018 8B 55 FC                      mov     edx, [ebp+var_4]
CODE:0058BEDB 018 8B 45 08                      mov     eax, [ebp+arg_0]    ; [16]
CODE:0058BEDE 018 8B 40 08                      mov     eax, [eax+8]        ; [17]
CODE:0058BEE1 018 8B 40 08                      mov     eax, [eax+8]        ; [18]
CODE:0058BEE4 018 8B 40 E8                      mov     eax, [eax-18h]      ; [19]
CODE:0058BEE7 018 8B 88 64 05 00 00             mov     ecx, [eax+564h]
CODE:0058BEED 018 8B 45 08                      mov     eax, [ebp+arg_0]
```

```
CODE:0058BEF0 018 8B 40 08                          mov     eax, [eax+8]
CODE:0058BEF3 018 8B 40 08                          mov     eax, [eax+8]
CODE:0058BEF6 018 8B 40 E8                          mov     eax, [eax-18h]
CODE:0058BEF9 018 E8 12 39 07 00                    call    sub_5FF810
CODE:0058BEFE 018 84 C0                             test    al, al
CODE:0058BF00 018 75 0C                             jnz     short loc_58BF0E
```

但是在使用IDA-minsc时，我们可以在参数中标记用于存储其调用者帧的函数，以及每个帧所属的函数地址。 这样我们就可以识别类似于[19]引用的指令的框架。
为此，我们将使用两个标记名称。 其中"frame-avar"用于存储包含调用者帧的参数名称，以及"frame-lvars"用于存储引用帧所属的函数地址。
我们可以参考图8，在地址0x590a32处，函数"sub_590728"将其帧作为参数传递给位于地址0x590a33处的调用指令。
我们可以通过双击它进入这个函数调用，之后IDA将定位到名为"sub_58BE98"的函数的最顶层。 此函数只有一个调用者，如果我们查看其引用（Ctrl + X）它将列出我们刚刚定位的地址。 知道这一点后，我们可以用它的调用方法的地址标记这个函数。

```
Python>callers = func.up()
Python>caller = callers[0]
Python>func.tag('frame-lvars', caller)
```

为了更容易识别参数，让我们使用"堆栈变量重命名"对话框将参数命名为"ap_frame_0"。 这可以通过选择"arg_0"然后点击"n"字符来完成此工作。
将变量重命名为"arg_0"后， 我们将再次使用标记将参数名称存储为"frame-avar"。 如果之后我们希望查找框参数，我们可以使用"frame-avar"标签来提取它。

```
Python>func.tag('frame-avar', 'ap_frame_0')
```

执行此操作后，该函数将如下图所示。 我们现在可以遍历"ap_frame_0"参数变量的任何引用，然后使用"frame-lvars"标记中的值标记它们。

```
CODE:0058BE98                       ; [frame-avar] ap_frame_0
CODE:0058BE98                       ; [frame-lvars] 0x590a33
CODE:0058BE98                       ; Attributes: bp-based frame
CODE:0058BE98
CODE:0058BE98                       sub_58BE98      proc near
CODE:0058BE98
CODE:0058BE98                       var_4           = dword ptr -4
CODE:0058BE98                       ap_frame_0      = dword ptr  8
CODE:0058BE98
```

为此我们将使用function.frame■■函数提取框架结构。 完成此操作后，我们可以获取表示"ap_frame_0"变量的成员。
然后，可以使用此结构成员枚举当前函数中对它的所有引用。

```
Python>f = func.frame()
Python>f.members
<type 'structure' name='$ F58BE98' size=+0x10>
[0] -4:+0x4 'var_4'      (<type 'int'>, 4)
[1]  0:+0x4 ' s'         [(<type 'int'>, 1), 4]
[2]  4:+0x4 ' r'         [(<type 'int'>, 1), 4]
[3]  8:+0x4 'ap_frame_0' (<type 'int'>, 4)
```

我们可以使用其索引或其名称来获得该成员变量。 在这种情况下，我们可以按名称引用它。
一旦获取了成员，我们就可以继续调用它的refs■■方法来遍历函数中对成员的所有引用。

```
Python>m = f.by('ap_frame_0')
Python>len(m.refs())
8
Python>for r in m.refs():
Python>    print r
Python>
AddressOpnumReftype(address=5815998L, opnum=1, reftype=ref_t(r))
AddressOpnumReftype(address=5816027L, opnum=1, reftype=ref_t(r))
AddressOpnumReftype(address=5816045L, opnum=1, reftype=ref_t(r))
AddressOpnumReftype(address=5816066L, opnum=1, reftype=ref_t(r))
AddressOpnumReftype(address=5816087L, opnum=1, reftype=ref_t(r))
AddressOpnumReftype(address=5816112L, opnum=1, reftype=ref_t(r))
AddressOpnumReftype(address=5816134L, opnum=1, reftype=ref_t(r))
AddressOpnumReftype(address=5816175L, opnum=1, reftype=ref_t(r))
```

返回的引用是一个命名元组，包含地址、操作数、以及引用是读取还是写入变量。
为了显示我们正在处理的指令，我们将简单地从元组中解压缩地址并将其与database.disassemble函数一起使用。

```
Python>for ea, _, _ in m.refs():
Python>    print db.disasm(ea)
Python>
58bebe: mov eax, [ebp+ap_frame_0]
58bedb: mov eax, [ebp+ap_frame_0]
```

```
58beed: mov eax, [ebp+ap_frame_0]
58bf02: mov eax, [ebp+ap_frame_0]
58bf17: mov eax, [ebp+ap_frame_0]
58bf30: mov eax, [ebp+ap_frame_0]
58bf46: mov eax, [ebp+ap_frame_0]
58bf6f: mov eax, [ebp+ap_frame_0]
```

现在我们可以确定这些指令已经成功引用了包含其调用帧的参数。让我们暂时用名称“frame-operand”和操作数索引标记它们。

```
Python>for ea, idx, _ in m.refs():
Python>    db.tag(ea, 'frame-operand', idx)
Python>
```

接下来，我们要做的是为每个使用分配帧变量的寄存器标识下一条指令。 我们可以使用`instruction.op`去识别属于第一个操作数的寄存器。
要找到正在读取寄存器的下一条指令，我们可以使用`database.address.nextreg`函数。
然而，在我们实际标记结果之前让我们首先选择“frame-operand”标记，并使用`instruction.op`和`database.address.nextreg`的组合来查看我们的结果。

```
Python>for ea, tags in func.select('frame-operand'):
Python>    reg = ins.op(ea, 0)
Python>    next_ref = db.a.nextreg(ea, reg, read=True)
Python>    print hex(ea), '->', db.disasm(next_ref)
Python>
58bebe -> 58bec1: push eax
58bedb -> 58bede: mov eax, [eax+8]
58beed -> 58bef0: mov eax, [eax+8]
58bf02 -> 58bf05: mov eax, [eax+8]
58bf17 -> 58bf1a: mov eax, [eax+8]
58bf30 -> 58bf33: mov eax, [eax+8]
58bf46 -> 58bf49: mov eax, [eax+8]
58bf6f -> 58bf72: mov eax, [eax+8]
```

在地址0x58bebe处我们能看到，■eax寄存器的下一次使用是通过“push”指令在0x58bec1处。 这可能用于将调用程序的帧传递给函数调用。
现在因为我们只对需要处理当前函数中使用的帧变量，所以我们将地址中的标记删除。

```
Python>db.tag(0x58bebe, 'frame-operand', None)
1
```

在删除地址标记后，这里应该只存在从帧中读取的赋值指令。 而之前我们已将调用者的地址存储在功能标记的“frame-lvars”中。
因此，我们现在可以使用它来标记每个赋值指令。

```
Python>for ea, tags in func.select('frame-operand'):
Python>    reg = ins.op(ea, 0)
Python>    next_ref = db.a.nextreg(ea, reg, read=True)
Python>    lvars = func.tag('frame-lvars')
Python>    db.tag(next_ref, 'frame', lvars)
Python>
```

现在我们已经创建了一个新的标签“frame”，它指向使用该帧的指令。此时我们不再需要“frame-operand”标签了。
我们现在可以通过在IDAPython命令提示符下执行以下代码来删除此“frame-operand”标记。

```
Python>for ea, _ in func.select('frame-operand'):
Python>    db.tag(ea, 'frame-operand', None)
Python>
```

让我们再次查看我们的结果，查询函数是否有任何标记为“frame”的指令。
我们将再次使用`database.disassemble`，这次操作使用“comment”关键字所指定的注释作为其参数之一。

```
Python>for ea, tags in func.select('frame'):
Python>    print db.disasm(ea, comment=True)
Python>
58bede: mov eax, [eax+8]; [frame] 0x590a33
58bef0: mov eax, [eax+8]; [frame] 0x590a33
58bf05: mov eax, [eax+8]; [frame] 0x590a33
58bf1a: mov eax, [eax+8]; [frame] 0x590a33
58bf33: mov eax, [eax+8]; [frame] 0x590a33
58bf49: mov eax, [eax+8]; [frame] 0x590a33
58bf72: mov eax, [eax+8]; [frame] 0x590a33
```

在用“frame”标记这些指令之后，我们现在可以看到偏移实际指的是哪个帧。 有了这个，当我们正在反汇编时，我们能够双击并立即查看拥有该变量的框架。
但是，我们可以做得比这更好一些。 如果我们双击我们发出的一个指令的地址，我们就可以使用`instruction.op`函数来提取引用该帧变量的操作数。
让我们定位到其中一个说明，然后尝试。

```
Python>ins.op(1)
OffsetBaseIndexScale(offset=8L, base=<internal.interface.register.eax(0,dt_dword) 'eax' 0:+32>, index=None, scale=1)
```

我们可以通过instruction.op立即发出当前指令的第一个操作数，并返回一个命名元组。它包含指向我们希望查看的帧变量的偏移量。
如果我们使用带有function.frame的偏移来识别框架，那么我们就可以得到它的命名。
让我们用这个方法获取成员名称然后用它作为“frame-member”标记指令。

```
Python>for ea, tags in func.select('frame'):
Python>    frame = func.frame(tags['frame'])
Python>    offset = ins.op(ea, 1).offset
Python>    member = frame.by(offset)
Python>    db.tag(ea, 'frame-member', member.name)
Python>
```

我们现在可以看到当前函数中标有“frame”的指令都包含frame变量名称的“frame-member”标记。
如果我们获得了任何用标签“frame”标记的指令，那么我们应先根据描述的代码查看由“frame”值标识的函数所拥有的帧，然后使用对其名称的引用对其进行标记。
这样，如果多个指令在标记中包含正确的帧，则先前的代码将存储被引用的变量名称。
如果我们在0x590a33确定函数的调用者，我们则可以对0x58bf36处的指令执行相同的操作。
有了这个，我们就可以用标签名称“frame”和函数的地址来标记0x58bf36。

```
CODE:0058BF33 018 8B 40 08                          mov     eax, [eax+8]    ; [frame] 0x590a33
CODE:0058BF33                                                               ; [frame-member] arg_0
CODE:0058BF36 018 8B 40 08                          mov     eax, [eax+8]
CODE:0058BF39 018 8B 80 54 F9 FF FF                 mov     eax, [eax-6ACh]
CODE:0058BF3F 018 8B D3                             mov     edx, ebx
```

然而我们可以不用使用标签来引用它。 IDA-minsc实际上允许我们通过instruction.op_structure函数将帧结构本身应用于操作数。
要执行此操作，我们将对“frame”标记执行相同的选择，而不是获取偏移量以确定框架成员的名称，我们可以使用带有框架结构本身的instruction.op_structure方法。

```
Python>for ea, tags in func.select('frame'):
Python>    frame = func.frame(tags['frame'])
Python>    ins.op_struct(ea, 1, frame)
Python>
```

然后每个标记的指令可以引用指向帧成员的第二个操作数。 下图将显示0x58bf49处的“frame”标记。

```
CODE:0058BF36 018 8B 40 08                          mov     eax, [eax+8]
CODE:0058BF39 018 8B 80 54 F9 FF FF                 mov     eax, [eax-6ACh]
CODE:0058BF3F 018 8B D3                             mov     edx, ebx
CODE:0058BF41 018 E8 EA BE E7 FF                    call    sub_407E30
CODE:0058BF46 018 8B 45 08                          mov     eax, [ebp+ap_frame_0]
CODE:0058BF49 018 8B 40 08                          mov     eax, [eax+($ F590728.arg_0-0Ch)] ; [frame] 0x590a33
```

总结

IDA-minsc中有许多功能，包括允许用户以编程的方式与IDA向逆向工程师所公开的二进制文件的各个部分进行交互。在我们上面提到的功能中，这个插件还包含“结构”模块

尽管与大多数逆向工程任务一样，很多事情都可以通过调试器和一些适当的断点来完成，但我们相信用户通过注释脚本能够编写兼容性很好的代码。通过使用一致的、非详细

请访问GitHub的存储库，下载该插件并进行试用。如果您喜欢该插件，请“关注”它，并在存储库中提出所发现的问题或贡献CONTRIBUTING.md文件。

上一篇：IDA-minsc在Hex-Ray... 下一篇：SSL证书真伪检测工具实现方法
1. 0 条回复
   - 动动手指，沙发就是你的了！

目录