IDA-minsc在Hex-Rays插件大赛中获得第二名（1）

Pinging / 2018-10-02 12:51:39 / 浏览数 3844 安全技术 技术讨论 顶(1) 踩(0)

■■■■■■■■■■■■
https://blog.talosintelligence.com/2018/09/ida-minsc.html?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+feedburne
■■■■■■■■■■■■■■■■■■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

介绍

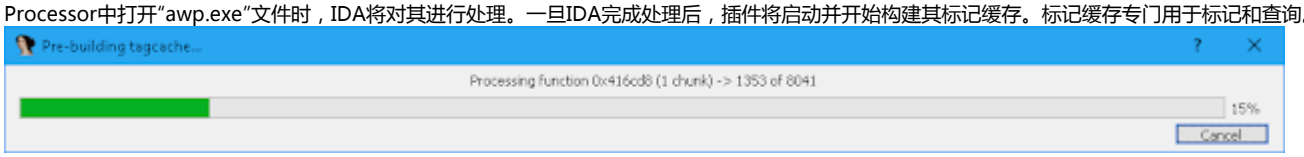
思科 Talos团队的Ali
Rizvi-Santiago使用名为"IDA-minsc"插件在IDA插件竞赛中获得第二名。IDA是由Hex-Rays公司创建的多处理器反汇编调试器，今年诞生了四名获奖者并提交了9个插件。
此插件旨在使人们更容易反汇编和注释二进制文件。我们相信此插件可加快注释过程并使用户更有效地工作。这是通过引入一些改变大多数用户开发Python方式的概念来完
这个功能与插件的各种组件相结合，可以根据用户当前的选择自动确定函数的参数，并允许用户快速编写用于标记和注释不同部分数据库的代码。
这个插件在这里有详细的文档介绍。下面，我们将通过反编译Atlantis Word Processor软件来展示这个插件的功能。而此软件是用Borland
Delphi编写的文档创建工具。本博客将概述如何快速标记查询对象，如何识别属于RTF解析器的标记及其属性，以及如何处理其他函数中定义的变量闭包。下面描述的所有功

背景

IDA Python本质上是IDA SDK的包装器，它使不同的IDA组件在被执行时有单独的模块直接与其相对应。由于IDA
6.95中使用的模块太过复杂，以至于用户无法对其熟悉。 于是IDA
Python通过使用更多更高级别的函数来快速解决这个问题。然而，新模块的命名过于通用化，并且需要以前的IDC脚本语言知识。
在编写这个新插件时，我们发现我们可以将各种组件和功能组合到单独的模块中，从而更容易调用、引用它们。这个插件中有各种模块，但其主要特征是IDA-minsc开始使用
这些模块都是具有静态方法的类定义，而这些静态方法用作"命名空间"以将那些作用于相似数据或语义的函数组合在一起。

创建初始化数据库

当首次在Atlantis Word
Processor中打开"awp.exe"文件时，IDA将对其进行处理。一旦IDA完成处理后，插件将启动并开始构建其标记缓存。标记缓存专门用于标记和查询。在此过程中，插件将选



标记所有类及其大小

所有Delphi应用程序通常都包含一个名为"TObject"的类。
这个类可以由其余许多类继承，并且我们可以利用它来查找通常用作构造函数的"System.New"。首先，我们使用IDA-minsc列出所有引用"TObject"的符号名称。
这与使用IDA的"名称"窗口（Shift + F4）相同，但使用IDA-minsc的匹配组件来指定不同的关键字以过滤IDA的不同窗口。

```
Python>db.names.list(like='*TObject')
   [    0] 0x4010b0 _cls_System_TObject
   [  382] 0x4058e4 GetCurrentObject
   [  408] 0x4059b4 SelectObject
   [11340] 0x67d658 __imp_SelectObject
   [11366] 0x67d6c0 __imp_GetCurrentObject
```

如果我们双击"_cls_System_TObject"的地址或符号，IDA将指向其指定的地址。 这将如下图1所示，它展示了我们的"TObject"。 如果我们进行相互参照（Ctrl + X）操作，我们可以看到这个字符串被它右边的0x401070地址引用。 这表示Delphi中的实际类定义。 该地址是继承了"TObject"的类的一个引用。

一旦我们有了这个地址，我们就可以再次获取它的引用来查看继承"TObject"的所有的类 - 看起来他们大约有122个。
如果我们随机选择一个，我们会看到其看起来像某种结构。 该结构以自引用开始，并且包括了许多可以调用的函数。
在Delphi中，此自引用用于区分用于控制对象范围和功能的默认函数。

如果我们跟随此引用，那么我们会发现它上面是与标准库相关的函数，而后面的函数是有关自定义实现的方法。

由于图2中的三个函数与标准库相关，因此我们可以看到，如果我们逐个查看它们它们会执行一些任务。
地址为0x406dec的第三个函数似乎调用了“CloseHandle”，所以这可能是析构函数。 地址为0x406De4的第一个函数通常是构造函数。
如果我们选择这个函数并列出被使用的引用（X），我们可以看到它共有473个。 我们将使用这些引用来查找每个类并标记它们。
然而，在我们这样做之前，让我们详细了解一下这个结构：

```
CODE:00406DAE 8B C0                                   align 10h
CODE:00406DB0 F0 6D 40 00           off_406DB0        dd offset off_406DF0    ; [1]
CODE:00406DB0
CODE:00406DB4 00 00 00 00 00 00+                      dd 7 dup(0)
CODE:00406DD0 F8 6D 40 00                             dd offset str.TThread   ; [3] "TThread"
CODE:00406DD4 28 00 00 00                             dd 28h                  ; [5]
CODE:00406DD8 70 10 40 00                             dd offset off_401070
CODE:00406DDC 38 2A 40 00                             dd offset sub_402A38
CODE:00406DE0 40 2A 40 00                             dd offset nullsub_9
CODE:00406DE4 98 28 40 00                             dd offset sub_402898    ; Constructor
CODE:00406DE8 AC 28 40 00                             dd offset sub_4028AC    ; Finalizer
CODE:00406DEC E8 A9 40 00                             dd offset sub_40A9E8    ; Destructor
CODE:00406DF0 38 AA 40 00           off_406DF0        dd offset loc_40AA38    ; [2]
CODE:00406DF4 C4 25 40 00                             dd offset nullsub_11
CODE:00406DF8 07 54 54 68 72 65+    str.TThread       db 7,'TThread'          ; [4]
CODE:00406E00 04 6E 40 00           off_406E00        dd offset byte_406E04
```

如上文所述，该结构在[1]中包含自引用。 因为该引用由数据库中的函数使用，所以它被IDA标记。
此引用专门用于访问[2]以找到类的“构造函数”、“终结符”和“析构函数”。 此外，在[3]的开头附近是一个指向字符串的指针。
该字符串表示位于[4]的类名及其内容。 此字符串的格式与Pascal中的格式相同，后者以单字节长度开头，后跟表示字符串的字节数。
最后，在[5]中有该类的长度。 这表示为了存储其成员而需要分配的大小。 首先，让我们快速定义一个在IDA Python命令行设置pascal样式字符串的函数。
为此，我们将使用`database.set.integer`命名空间将第一个字节作为长度的`uint8_t`。
对于字符串的其余部分，我们将使用带有长度的`database.set.string`将地址转换为指定长度的字符串。

```
Python>def set_pascal_string(ea):
Python>    ch = db.set.i.uint8_t(ea)
Python>    return db.set.string(ea + 1, ch)
```

完成后，如果我们需要我们可以使用`database.get.string`来读取它。
尽管`database.set`命名空间返回的是已创建的值，但我们可以使用以下内容对上面指定的代码进行反编译。

```
Python>def get_pascal_string(ea):
Python>    ch = db.get.i.uint8_t(ea)
Python>    return db.get.string(ea + 1, length=ch)
```

现在我们可以输入以下内容用来读取该地址的字符串

```
Python>print get_pascal_string(0x406df8)
TThread
```

现在我们可以获取并应用那些可以为我们提供名称的字符串，并且我们可以使用此名称来标记该类。
为此，我们将使用构造函数的所有引用，并使用每个引用来计算所有类的不同字段。
之后，我们将使用标签来标记不同的对象，以便我们以后可以在必要时进行查询。 首先，我们首先双击地址为0x406de4的“构造函数”。
这会将我们带我们来到函数“sub_402898”处。 既然我们知道这个函数是什么，那就让我们用以下命名：

```
Python>func.name('System.New')
sub_402898
```

如果你注意到，我们并没有提供地址。 因为没有提供地址，所以我们假设使用当前函数。 这是IDA-minsc的“多功能”组件。
如果我们对`function.name`运行`help`■■，我们可以看到其他变量：

```
Python>help(function.name)
Help on function name in module function:
name(*arguments, **keywords)
```

```
name() -> Return the name of the current function.
name(string=basestring, *suffix) -> Set the name of the current function to ``string``.
name(none=NoneType) -> Remove the custom-name from the current function.
name(func) -> Return the name of the function ``func``.
name(func, none=NoneType) -> Remove the custom-name from the function ``func``.
name(func, string=basestring, *suffix) -> Set the name of the function ``func`` to ``string``.
```

现在我们已经命名了这个函数，我们将遍历它的所有引用并获取它的不同字段。 作为我们上面描述的字段的参考，我们获得了以下内容：

```
CODE:00406DAE 8B C0                                   align 10h
CODE:00406DB0 F0 6D 40 00           off_406DB0        dd offset off_406DF0    ; [6] Top of class or Info (Reference - 16*4)
CODE:00406DB0
CODE:00406DB4 00 00 00 00 00 00+                      dd 7 dup(0)
CODE:00406DD0 F8 6D 40 00                             dd offset str.TThread   ; [7] Class name (Reference - 8*4)
CODE:00406DD4 28 00 00 00                             dd 28h                  ; [8] Class size (Reference - 7*4)
CODE:00406DD8 70 10 40 00                             dd offset off_401070    ; [9] Parent class (Reference - 6*4)
CODE:00406DDC 38 2A 40 00                             dd offset sub_402A38
CODE:00406DE0 40 2A 40 00                             dd offset nullsub_9
CODE:00406DE4 98 28 40 00                             dd offset sub_402898    ; [10] Constructor (Reference - 3*4)
CODE:00406DE8 AC 28 40 00                             dd offset sub_4028AC    ; [11] Finalizer (Reference - 2*4)
CODE:00406DEC E8 A9 40 00                             dd offset sub_40A9E8    ; [12] Destructor (Reference - 1*4)
CODE:00406DF0 38 AA 40 00           off_406DF0        dd offset loc_40AA38    ; [13] * Reference
CODE:00406DF4 C4 25 40 00                             dd offset nullsub_11
CODE:00406DF8 07 54 54 68 72 65+    str.TThread       db 7,'TThread'
CODE:00406E00 04 6E 40 00           off_406E00        dd offset byte_406E04
```

通过这种布局，我们可以提取引用构造函数的所有类的不同组件，并标记它们以便稍后进行查询。
因为早期我们双击构造函数并命名它，所以我们当前应该在"System.New"函数中。 要获取所有引用值，我们可以使用function.up■■方法。
然后我们将遍历其所有引用，添加0xc（3 * 4 == 12）以获得[13]处的引用，然后使用它来定位其余字段。
对于类名[7]，我们将使用我们的set_pascal_string和get_pascal_string函数。对于标准作用域构造[10]，[11]和[12]，我们将对其进行操作并用它们标记其"类型"。 这导致以下代码。 以下代码可以做得更短，但为了便于阅读所以我们进行了扩展。

```
Python>for ea in func.up():
Python>    ref = ea + 3*4  # [13] calculate address to reference
Python>
Python>    # read our fields
Python>    lookup = {}
Python>    lookup['info'] = ref - 16*4         # [6]
Python>    lookup['name'] = ref - 8*4          # [7]
Python>    lookup['size'] = ref - 7*4          # [8]
Python>    lookup['parent'] = ref - 6*4        # [9]
Python>    lookup['constructor'] = ref - 3*4   # [10]
Python>    lookup['finalizer'] = ref - 2*4     # [11]
Python>    lookup['destructor'] = ref - 1*4    # [12]
Python>    lookup['object'] = ref              # [13]
Python>
Python>    # dereference any fields that need it
Python>    name_ea = db.get.i.uint32_t(lookup['name'])
Python>    parent_ea = db.get.i.uint32_t(lookup['parent'])
Python>    size = db.get.i.uint32_t(lookup['size'])
Python>
Python>    # set our name (just in case IDA has it defined as something else)
Python>    set_pascal_string(name_ea)
Python>
Python>    # decode our name
Python>    name = get_pascal_string(name_ea)
Python>
Python>    # name our addresses
Python>    db.name(lookup['info'], 'gv', "Info({:s})".format(name))
Python>    db.name(lookup['object'], 'gv', "Object({:s})".format(name))
Python>
Python>    # tag our methods
Python>    m_constructor = db.get.i.uint32_t(lookup['constructor'])
Python>    func.tag(m_constructor, 'function-type', 'constructor')
Python>    m_finalizer = db.get.i.uint32_t(lookup['finalizer'])
Python>    func.tag(m_finalizer, 'function-type', 'finalizer')
Python>    m_destructor = db.get.i.uint32_t(lookup['destructor'])
Python>    func.tag(m_destructor, 'function-type', 'destructor')
Python>
Python>    # tag our class structure
```

```
Python>    db.tag(lookup['info'], 'object.name', name)
Python>    db.tag(lookup['info'], 'object.methods', lookup['object'])
Python>    db.tag(lookup['info'], 'object.size', size)
Python>    if parent_ea:
Python>        db.tag(lookup['info'], 'object.parent', parent_ea)
Python>    continue
```

这将导致数据库中的所有Delphi对象都被标记。
由于标签利用的是数据库中的注释，因此当用户正在逆向时，他们可以立即看到与给定地址关联的标签可能的样子。 在IDA Python命令行执行了上一个代码后，“TThread”对象的外观如图3所示。

在此大块代码被执行后，数据库中的每个对象应该被标记。 这将允许我们使用database.select■■查询数据库，以便找到特定大小的类。
有关更多信息，请查看database.select中的help()函数。 使用它来查找0x38的对象大小的示例如下：

```
Python>for ea, tags in db.select(Or=('object.name', 'object.size')):
Python>    if tags['object.size'] == 0x38:
Python>        print hex(ea), tags
Python>    continue
```

记录RTF的token值

Atlantis Word Processor包含RTF解析器。 这种文件格式是众所周知的，因此很容易识别它支持的标记，并希望找到负责解析每个标记参数的函数。
为此，我们将首先搜索定义在数组中的“objemb”字符串。 首先，我们可以使用IDA的文本搜索（Alt + T）。但我们可以通过使用IDA-minsc中database.search命名空间提供的功能，并与go■■函数结合。如此以来，我们便可以立即导航到制定内容。

```
Python>go(db.search.by_text('objemb'))
```

下图为将我们直接定位到“objemb”字符串的第一个实例。 如果我们进行对照（Ctrl + X），我们便能发现只有一个使用它的引用。
这将我们带到图4中的字符串引用列表。

由于没有引用，我们可以快速导航到IDA之前定义的标签。 由于IDA具有引用该特定地址的反汇编代码，因此该标签必然存在。
为此，我们可以使用database.address命名空间中的功能。 包括nextlabel■■和prevlabel■■函数。
我们可以通过在IDAPython命令行输入以下内容来使用它们：

```
Python>go(db.a.prevlabel())
```

这个数据似乎是一个数组，但IDA还没有对其进行定义。 我们可以点击'*'来调出IDA的“转换为数组”对话框，但我们可以使用IDA-minsc来代替。
我们将使用database.address.nextlabel■■和database.here■■（别名为h■■）来计算数组的元素数，然后使用databaset.set.array()将它分配到数据库中。
database.set.array函数采用“pythonic”类型作为其参数之一。
这在IDA-minsc的文档中进行了描述，其允许我们在IDA中描述类型，而无需理解正确的标志或类型。
在这种情况下，我们可以使用■int■4■来指定一个四字节整数（dword），但由于这是一个32位数据库，我们可以使用int来使用默认的整数大小。

```
Python>size = (db.a.nextlabel() - h())
Python>db.set.array(int, size / 4)
```

我们也使用database.name■■命名当前地址的数组：

```
Python>db.name('gv', "rtfTokenArray({:d})".format(db.t.array.length()))
```

看起来这个数组的某些元素没有被IDA标记为实际的字符串，如图5所示：

我们可以通过遍历此数组中的所有地址来取消定义地址，然后将其重新定义为字符串，以与手动操作相同的方式快速修复此问题。
这可以通过IDAPython命令行中的以下内容完成：

```
Python>for ea in db.get.array():
Python>    db.set.undefined(ea)
Python>    db.set.string(ea)
```

现在我们修复了这个数组。 如果我们导航回到顶部，我们会注意到这个数组与许多数组是连续的。
让我们首先修复这个数组并将我们当前的位置保存在变量position中，然后使用database.address.prevlabel■■来实现。
如此以来，我们可以像处理第一个数组那样进行其他操作。

```
Python>position = h()
Python>
Python>go(db.a.prevlabel())
Python>
Python>db.set.array(int, (db.a.nextlabel() - h()) / 4)
Python>db.name('gv', "rtfTokenArray({:d})".format(db.t.array.length()))
Python>for ea in db.get.array():
Python>    db.set.undefined(ea)
Python>    db.set.string(ea)
```

现在我们可以返回到之前保存的位置并重复操作一下两个数组：

```
Python>go(position)
Python>
Python>go(db.a.nextlabel())
Python>
Python>db.set.array(int, (db.a.nextlabel() - h()) / 4)
Python>db.name('gv', "rtfTokenArray({:d})".format(db.t.array.length()))
Python>for ea in db.get.array():
Python>    db.set.undefined(ea)
Python>    db.set.string(ea)
Python>
Python>go(db.a.nextlabel())
Python>
Python>db.set.array(int, (db.a.nextlabel() - h()) / 4)
Python>db.name('gv', "rtfTokenArray({:d})".format(db.t.array.length()))
Python>for ea in db.get.array():
Python>    db.set.undefined(ea)
Python>    db.set.string(ea)
```

现在它已经完成，我们可以列出我们使用database.names命名空间创建的所有数组。 让我们列出以“gv_rtfToken”开头的所有符号。
从这个列表中，让我们看一下我们定义的第一个数组（“gv_rtfTokenArray（213）”），然后双击它的地址。

```
Python>db.names.list('gv_rtfToken*')
[11612] 0x668ba8 gv_rtfTokenArray(64)
[11613] 0x668ca8 gv_rtfTokenArray(213)
[11614] 0x668ffc gv_rtfTokenArray(46)
[11615] 0x6690b4 gv_rtfTokenArray(135)
```

现在我们应该定义“gv_rtfTokenArray（213）”。 如果我们进行对照（Ctrl + X），我们可以看到在地址0x431DD7（[14]）处只有一个代码引用。

```
CODE:00431DD7 000 A1 A8 8C 66 00                    mov     eax, ds:gv_rtfTokenArray(213)   ; [14]
CODE:00431DDC 000 A3 EC 85 67 00                    mov     ds:dword_6785EC, eax            ; [15]
CODE:00431DE1 000 C7 05 F0 85 67 00+                mov     ds:dword_6785F0, 4
CODE:00431DEB 000 C7 05 F4 85 67 00+                mov     ds:dword_6785F4, 4
CODE:00431DF5
CODE:00431DF5                       locret_431DF5:
CODE:00431DF5 000 C3                                retn
CODE:00431DF5                       sub_431D38      endp
```

该指令读取token数组地址，然后将其写入另一个全局[15]中。 因为这只是指向我们数组的指针，所以我们也要命名这个地址。
而不是使用IDA的“重命名地址”对话框，或双击“dword_6785EC”并使用带有当前地址的database.name。我们实际上是直接从指令的操作数中提取出地址。
这可以通过instruction.op函数完成。 如果我们选择地址0x431ddc，我们的全局token数组将驻留在当前指令的第一个操作数中。
我们可以在IDAPython命令行中将其操作数作为命名元组提取：

```
Python>ins.op(0)
OffsetBaseIndexScale(offset=6784492L, base=None, index=None, scale=1)
```

由于我们没有提供地址作为instruction.op的第一个参数，因此我们假设我们处理的是当前操作。 命名元组的“offset”字段包含我们的dword的地址，
因此我们可以使用以下命令使用已选定的相同地址进行命名。 由于此地址已经具有名称“dword_6785EC”，database.name函数将返回原始名称。

```
Python>ea = ins.op(0).offset
Python>db.name(ea, 'gp','rtfTokenArray(213)')
dword_6785EC
```

同一个函数对我们之前定义的所有数组的全局指针进行相同的赋值。 我们可以重复此过程来命名所有这些，然后交叉引用它们以找到RTF标记生成器。
现在，让我们在达到这一点之前做好准备。 我们的准备工作只需要回到我们的令牌数组并从中提取字符串。
我们已经有了这些命名，所以我们可以列出以下内容：

```
Python>db.names.list('gv_rtfToken*')
[11612] 0x668ba8 gv_rtfTokenArray(64)
[11613] 0x668ca8 gv_rtfTokenArray(213)
[11614] 0x668ffc gv_rtfTokenArray(46)
[11615] 0x6690b4 gv_rtfTokenArray(135)
```

然而我们想要遍历此列表。 database.names命名空间包含一个专门用于此目的iterate■■函数。
我们可以将它与database.get.array■■结合使用，用于将数组存储为单个列表。 在IDAPython命令行，我们将执行以下操作：

```
Python>tokens = []
Python>for ea, name in db.names.iterate('gv_rtftoken*'):
Python>    rtfTokenArray = db.get.array(ea)
Python>    tokens.extend(rtfTokenArray)
Python>
Python>len(tokens)
458
```

我们有一个包含458个地址并指向实际的RTF令牌的列表。 我们将使用快速列表解析将其转换为字符串列表，以将地址映射到字符串。
现在我们可以将令牌标识符转换为其实际的令牌字符串。

```
Python>tokens = [db.get.string(ea) for ea in tokens]
```

■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

上一篇：Java沙箱逃逸走过的二十个春秋（一） 下一篇：IDA-minsc在Hex-Ray...

1. 0 条回复
   - 动动手指，沙发就是你的了！

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板