Python内存工具

1. 前言

在对G94项目进行服务器端内存泄漏分析和客户端外挂工具学习时，走了不少弯路也踩了不少坑，但是在这个过程中还是发现了不少好用的python工具的。在这里总结分享一波。

- 本文主要列举了python内存修改和内存泄漏分析的一些通用工具
- 对python内存修改工具memorpy中存在的"坑"进行的修复和优化进行了阐述
- 最后讲述了如何使用内存工具，对服务器内存泄漏进行定位和分析思路和方法

2. 工具与原理

以下是本人使用过的工具，在每个工具的说明中会分别阐述工具的原理、坑点和优化方法等，希望对大家有用。

本人对改进的工具进行了封装：http://git-qa.gz.netease.com/shiwei/mem_tools

2.1 Python内存读取/修改工具的原理

应用程序被操作系统加载后，操作系统会给应用程序分配内存，每个程序所看到的内存都是虚拟内存，虚拟内存的的起始地址都是一样的（Windows的装载基址都是0x00400000，感兴趣推荐看一下黑客相关的书《0day》）。只要得到了某个变量所在的虚拟内存地址，就能修改这个变量的值。修改内存时需要注意一下几点：

- Python对象的虚拟地址可以直接使用id内置函数获取
- 不同操作系统修改内存值的方法是不一样的
  - Windows利用kernel32.dll中的API
  - Linux实现了POSIX标准，所以只要通过attach相关进程，然后利用Python的ctypes调用C的接口
- 修改Python内存变量时需要知道变量的数据结构

```
// str对象的内存结构
// 如果想修改一个python字符串的内存实际上需要修改ob_sval所在的内存，由于32bit和64bit的操作系统
// 的long和int占用的字节数不同，所以需要跳过的字节也不同
typedef struct {
    PyObject_VAR_HEAD
    long ob_shash;
    int ob_sstate;
    char ob_sval[1];
} PyStringObject;
```

- 不同类型操作系统存储的数据可能是大端序可能是小端序，必要时需要进行转换

```
# 例如：一个整形数在Linux和Windows上的不同
0xdeadbeef # 4字节的整形数
de ad be ef # windows
ef be ad de # linux
```

ps. 上面例子中的deadbeef其实是挺好玩的东西，这个是操作系统默认填充已分配未使用的内存，Linux填充deadbeef，Windows填充0xCCCCCCCC，所以在Windows上读取的内存地址越界时输出的都是“烫”，那是因为“烫”字的unicode编码是0xCCCCCC

2.2 内存修改工具：memorpy

memorpy是一款在github上找到的内存修改工具（https://github.com/n1nj4sec/memorpy），功能十分强大，实现了Windows、Linux和Mac os上的内存修改方法。详细的使用方法在github的项目主页上有介绍，下面说一下工具的常用API，以及本人使用这个工具时遇到的坑和修复方法。

2.2.1 常用API和坑

- MemWorker
  - mem_search，对内存的搜索，可以使用正则表达式进行搜索，原理是将对应程序的所有内存值取出进行对比，如果满足则输出
  - parse_float_function，转换float数据字节码，搜索float效率非常低，对4096B的内存进行查找需要0.05s
    - 原理：使用struct先pack出float类型的数值，然后对比int的部分是否一样
    - 优化：快速修改float的方法是使用字符串替换的方式，例如：对某个变量a = 1000.0进行内存修改是，先使用struct对a转换成字节码，在对查找到的字节码进行替换（下面的例子主要针对32bit python的，64bit的需要的了解的朋友可以联系我，NeoX引擎用的python的32bit的）
    - 扩展：对于其他类型的数值修改，比如：int，可以先使用sys.getsizeof(1)拿到python实现int的size大小，再通过下面两个函数查找对应byte进行修改

```
# 拿到需要的浮点数的byte数组，python的byte数组是24位的
def get_float_bytes_template(fvalue):
    import os
    process = memorpy.Process(pid=os.getpid())
    return process.read_bytes(id(fvalue), bytes=16)
# 通过只对比后8个byte的方式去实现查找，因为CPython的实现方法是head+double的方式
def modify_python_float(pid, old_value, new_value):
    """
    由于前8个Byte是head，所以应该用double去找就可以了
    typedef struct {
        PyObject_HEAD
        double ob_fval;
    } PyFloatObject;
    :return:
    """
    mw = memorpy.MemWorker(pid=pid)
    tmp_bytes = get_float_bytes_template(old_value)
    tmp_bytes = tmp_bytes[8:] # 去掉前8个Byte的header
    ret = [x for x in mw.mem_search(tmp_bytes)]
    for r in ret:
        r.write(get_float_bytes_template(new_value)[8:])
```

  - parse_any_function，将需要查找的内存结构作为一个字符串进行查找，使用str的find函数，搜索效率还可以

```
mw = memorpy.MemWorker(pid=pid)
ret = [x for x in mw.mem_search('uuummm012')]
for r in ret:
    print r.write('mmmuuu123')
```

- WinProcess
  - 一个windows的进程对象
  - 在64bit机器上运行的程序的虚拟地址空间都是从0到2^31-1的，而0-65535和2^31-1-65535的地址都是操作系统调度用的，所以一个程序的地址空间的读取范围在65536-2147418111的地址空间中
  - iter_region，罗列一个进程的所有内存块
  - VirtualQueryEx，读取某个地址（int32）处的地址的存储信息
- MEMORY_BASIC_INFORMATION

```
class MEMORY_BASIC_INFORMATION(Structure):
    _fields_ = [('BaseAddress', c_void_p),
     ('AllocationBase', c_void_p),
     ('AllocationProtect', DWORD),
     ('RegionSize', c_size_t),
     ('State', DWORD),
     ('Protect', DWORD),
     ('Type', DWORD)]
```

这里主要关心一下数据结构：

  - BaseAddress，地址信息，如：65536
  - Protect，是否被保护，如：1表示受保护内存区域
  - RegionSize，当前内存块的大小，从BaseAddress+RegionSize为一个完整的内存数据
  - State，当前内存的状态信息，为65536表示空闲内存，为8192表示系统预留内存

2.2.2 memorpy库在Linux上无法通过pid attach进程

- 通过C实现读写当前进程上的内存
- 实现方法如下，贴两段C读写内存的方法，具体代码可以到git上获取
- http://git-qa.gz.netease.com/shiwei/mem_tools.git

```
int write(size_t p, char *data, int length) {
    char *pdata = (char*) p; //这里传入的p是需要写入的变量内存地址
    for (int i = 0; i < length; i++) {
        pdata[i] = (char) data[i];
    }
    return length;
}

m_type *read(size_t p, int length) {
    char *pdata = (char*) p; //id内置函数返回的int数即为变量所在读取的内存地址
    m_type *data = (m_type*) malloc(sizeof(m_type) * length);
    for (int i = 0; i < length; i++) {
        data[i] = (m_type)pdata[i];
    }
    return data;
```

```
}
```

2.3 内存泄漏跟踪工具：tracemalloc

- 在脚本层面引入tracemalloc库，tracemalloc在python3之后自带在库中，python2需要自己额外编译接入（教程：https://pytracemalloc.readthedocs.io/install.html#patch-python），tracemalloc可以统计、跟踪和分析脚本层面的内存分配情况，tracemalloc的文档：https://docs.python.org/3/library/tracemalloc.html
- G94项目接入后的使用方法：http://km.netease.com/article/259769

项目内已经配置好tracemalloc了，通过运行设计好的测试用例得到了可能存在的内存泄漏具体位置

```
1. ./game/systems/safe_zone.py:108: size=3660 KiB (+3660 KiB), count=3786 (+3786), average=990 B
      File "./game/systems/safe_zone.py", line 108 relative traceback:
        3786 memory blocks: 3660.0 KiB
          File "./game/systems/safe_zone.py", line 108
      File "./game/systems/safe_zone.py", line 108 total:  3748567
2. ./game/systems/physics.py:28: size=647 KiB (+647 KiB), count=966 (+966), average=686 B
      File "./game/systems/physics.py", line 28 relative traceback:
        966 memory blocks: 647.0 KiB
          File "./game/systems/physics.py", line 28
      File "./game/systems/physics.py", line 28 total:  663194
3. ./game/systems/state_input.py:26: size=227 KiB (+227 KiB), count=235 (+235), average=992 B
      File "./game/systems/state_input.py", line 26 relative traceback:
        235 memory blocks: 227.0 KiB
          File "./game/systems/state_input.py", line 26
      File "./game/systems/state_input.py", line 26 total:  233120
4. ./game/entity/component_physics.py:68: size=107 KiB (+107 KiB), count=1555 (+1555), average=71 B
      File "./game/entity/component_physics.py", line 68 relative traceback:
        1555 memory blocks: 107.0 KiB
          File "./game/entity/component_physics.py", line 68
      File "./game/entity/component_physics.py", line 68 total:  110496
5. ./game/entity/component_state_player.py:93: size=95.0 KiB (+95.0 KiB), count=99 (+99), average=992 B
      File "./game/entity/component_state_player.py", line 93 relative traceback:
        99 memory blocks: 95.0 KiB
          File "./game/entity/component_state_player.py", line 93
      File "./game/entity/component_state_player.py", line 93 total:  98208
```

2.4 内存对象分析工具：GC

GC提供了一个可选的垃圾回收接口。它能够被关掉，被调节垃圾回收频率和调试。它提供程序已经无法访问但未被释放的对象的访问接口。可以使用gc.set_debug(gc.DEBUG_LEAK)来调试一个已经存在泄漏的程序，开启debug模式后GC会将所有收集的对象都保存到gc.garbage用于检查。

- gc.isenabled()，判断是否开启自动回收垃圾
- gc.collect([generation])，默认对python的所有内存进行一次回收，可以指定0,1,2分别回收新生代、老生代、持久代的内存进行回收，函数返回有多少程序无法访问但未释放的对象数量
- gc.get_objects()，获得所有被track的objects，本身返回的列表不在其中
- gc.set_threshold(threshold0, threshold1,threhold2)，threshold(x，x>0)表示当generation(x-1)被收集thresholdx次后generationx开始被收集，threshold0则表示当allocations-deallocations>threshold0时generation0的收集被执行
- gc.get_count()，返回三个代上的对象数量
- gc.getreferrers(*obj)，返回所有引用了obj的对象，如果不想看到一些存在引用环而保留在内存中的对象的引用，则先调用gc.collect()进行收集一发，除了调试在开发中不要使用
- gc.getreferents(*obj)，返回所有被obj引用的对象列表

2.5 struct

Python的struct库十分强大，可以高效的完成Python和C的内存交互

- struct入门教程：https://www.cnblogs.com/gala/archive/2011/09/22/2184801.html

- struct官方文档：https://docs.python.org/2/library/struct.html

3. 内存泄漏分析方法

1. 分析并根据实际情况设计测试用例，G94（大逃杀）一个玩家完成一场游戏前后，内存应该不存在泄漏
2. 通过tracemalloc记录并分析存在的内存泄漏情况，下图结合了代码分析tracemalloc报出的内存泄漏代码节点

```
server reversion: 89792
###################################
import配置数据
1. ./game/systems/safe_zone.py:108: size=3660 KiB (+3660 KiB), count=3786 (+3786), average=990 B
      File "./game/systems/safe_zone.py", line 108 relative traceback:
        3786 memory blocks: 3660.0 KiB
          File "./game/systems/safe_zone.py", line 108
      File "./game/systems/safe_zone.py", line 108 total:  3748567
###########################################
import配置数据
2. ./game/systems/physics.py:28: size=647 KiB (+647 KiB), count=966 (+966), average=686 B
      File "./game/systems/physics.py", line 28 relative traceback:
        966 memory blocks: 647.0 KiB
          File "./game/systems/physics.py", line 28
      File "./game/systems/physics.py", line 28 total:  663194
###########################################
有可能存在内存泄漏
3. ./game/systems/state_input.py:26: size=310 KiB (+310 KiB), count=321 (+321), average=992 B
      File "./game/systems/state_input.py", line 26 relative traceback:
        321 memory blocks: 310.0 KiB
          File "./game/systems/state_input.py", line 26
      File "./game/systems/state_input.py", line 26 total:  318432
###########################################
```

3. 在可疑的存在内存泄漏的地方（例如上图的第3个地方）打上log，先确认是否存在泄漏问题，有可能工具误报（这个还不知道为什么），根据运行后的log并通过mem_tools工具读取泄漏的对象的值，如果并没有得到log中的值说明并不存在泄漏

```
log.Trace("mem_leak_log", "test_value", test_value, "type",
type(test_value).__name__)
```

4. 如果仍然存在泄漏，则通过GC分析当前内存对象的索引图，一般通过变量值和泄漏的具体位置，程序就能分析出泄漏的原因了

写在最后

联系方式：shiwei@corp.netease.com