Injector

علی اشتهاری پور

پایان نامه لیسانس در زمینه ی API Hooking

استاد راهنما: دکتر سپهر کیخایی

دانشگاه سپاهان

12/12/2013

در ویندوز هر پروسه فضای آدرس مجازی خصوصی خود را دریافت می کند. زمانی که یک اشاره گر به آدرسی اشاره می کند، در واقع این آدرس در این فضای آدرس قابل دسترسی است و نمی توان به فضای آدرس پروسه ی دیگری اشاره کرد.

حال مواردی پیش می آید که باید این مرزها را شکست و به فضای آدرس پروسه های دیگر دسترسی پیدا کرد:

- زمانی که دسترسی به پنجره ی پروسه ی دیگری نیاز باشد.
- زمانی که عیب یابی مطلوب باشد.
- زمانی که در تله (Hook) انداختن دیگر پروسه ها مورد نیاز باشد.

در این سند، تکنیک در تله انداختن پروسه بررسی می شود.

برای اجرای روالی دلخواه در پروسه ای دیگر، باید آن روال را در یک کتابخانه پویا (DLL) پیاده کرد، سپس آن کتابخانه را در آن پروسه تزریق (DLL Injection) نمود.

برای تزریق کتابخانه روش های متفاوتی وجود دارد:

- تزریق با استفاده از Registry.
- تزریق با استفاده از تله های ویندوز.
- تزریق با استفاده از نخ های دور (Remote Threads).
- تزریق یک کتابخانه ی تروجان.
- تزریق با استفاده از روش های عیب یابی.

تزریق با استفاده از نخ های دور بیشترین انعطاف پذیری را به ارغمان می آورد.

Remote Threads

این تکنیک نیازمند آن است که پروسه ی هدف روال LoadLibrary را فراخوانی کند تا کتابخانه در پروسه ی مقصد بارگزاری شود.

ویندوز تابعی را در اختیار توسعه دهندگان قرار داده که با استفاده از آن می توان نخی را در پروسه ی دیگری اجرا نمود.

```
HANDLE CreateRemoteThread(
   HANDLE hProcess,
   PSECURITY_ATTRIBUTES psa,
   DWORD dwStackSize,
   PTHREAD_START_ROUTINE pfnStartAddr,
   PVOID pvParam,
   DWORD fdwCreate,
   PDWORD pdwThreadId);
```

آرگومان hProcess در واقع دستگیره ای به آن پروسه ای است که باید نخ در آن اجرا شود.

آرگومان pfnStartAddr آدرسی است که نخ باید آن را اجرا کند. این آدرس یقینا در فضای آدرس پروسه ی هدف است.

برای آن که پروسه ی هدف کتابخانه ی مورد نظر را فراخوانی کند، باید با استفاده از این نخ، روال LoadLibrary را اجرا نمود.

روال مورد نظر در فایل سرآیند WinBase.h به این صورت بسته بندی شده است:

```
HMODULE WINAPI LoadLibraryA(LPCSTR lpLibFileName);
HMODULE WINAPI LoadLibraryW(LPCWSTR lpLibFileName);
#ifdef UNICODE
#define LoadLibrary LoadLibraryW
#else
#define LoadLibrary LoadLibraryA
#endif // !UNICODE
```

می توان مشاهده نمود که اگر از رمزگذاری ANSI استفاده شود، باید از LoadLibraryA و اگر از UNICODE استفاده شود، باید از LoadLibraryW استفاده نمود. نوع آرگومان ارسالی به این دو تابع از نظر رمزگذاری کاراکتر متفاوت می باشد.

بنابراین تمام کاری که مورد نیاز است انجام شود آنست که روال زیر فراخوانی شود:

```
HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0,
   LoadLibraryW, L"C:\\MyLib.dll", 0, NULL);
```

و یا اگر استفاده از ANSI ترجیح داده می شود:

```
HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0,
   LoadLibraryA, "C:\\MyLib.dll", 0, NULL);
```

زمانی که نخ جدید در پروسه ی هدف اجرا می شود، سریعا LoadLibraryW یا LoadLibraryA را با ارسال آرگومان مسیر کتابخانه به روال، فراخوانی می کند. در این جا دو مشکل وجود دارد.

اولین مشکل آن است که نمی توان به سادگی روال LoadLibraryA را عنوان آرگومان به CreateRemoteThread ارسال کرد.

دلیل آن بسیار دقیق است. زمانی که یک برنامه کامپایل و لینک می شود، باینری حاصل بخشی به نام واردات (import) دارد.

این بخش شامل قسمت هایی حاوی "توابع وارد شده" می باشد. پس زمانی که در کد برنامه روال LoadLibraryA فراخوانی می شود، الحاق دهنده (Linker)، فراخوانی ای را تولید می کند که در بخش اشاره شده موجود است.

اگر ارجاع مستقیمی به LoadLibraryA به CreateRemoteThread ارسال شود، در واقع آدرس این روال در پروسه ی مرجع به آن ارسال می شود. این در حالیست که پروسه ی هدف به فضای آدرس پروسه ی مرجع دسترسی ندارد. بنابراین معلوم نیست که نخ مورد نظر چه قطعه کدی را اجرا خواهد کرد. بنابراین باید با فراخوانی GetProcAddress آدرس دقیق این روال را بدست آورد.

روال CreateRemoteThread انتظار دارد که کتابخانه Kernel32.dll در فضای آدرس یکسانی در پروسه ی مرجع و هدف نگاشت شده باشد. همه ی برنامه های کاربردی به این کتابخانه نیاز دارند، پس انتظار می رود که انتظارات روال مورد نظر بدون تغییراتی، برآورده شده باشد.

همچنین روال LoadLibraryA و یا LoadLibraryW در کتابخانه Kerlen32.dll موجود می باشد بنابراین باید آدرس مطلق را از این کتابخانه بدست آورد.

بنابراین باید این گونه عمل کرد:

```
// Get the real address of LoadLibraryW in Kernel32.dll.
PTHREAD_START_ROUTINE pfnThreadRtn = (PTHREAD_START_ROUTINE)
   GetProcAddress(GetModuleHandle(TEXT("Kernel32")),
"LoadLibraryW");

HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0,
   pfnThreadRtn, L"C:\\MyLib.dll", 0, NULL);
```

و یا اگر استفاده از ANSI ترجیح داده می شود:

```
// Get the real address of LoadLibraryA in Kernel32.dll.
PTHREAD_START_ROUTINE pfnThreadRtn = (PTHREAD_START_ROUTINE)
   GetProcAddress(GetModuleHandle(TEXT("Kernel32")),
"LoadLibraryA");

HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0,
                              pfnThreadRtn, "C:\\MyLib.dll", 0, NULL);
```

مشکل دوم آنست که رشته ی C:\\MyLib.dll در فضای آدرس پروسه ی مرجع تعریف شده است و نمی توان از آن در پروسه ی هدف استفاده کرد.

برای تصحیح این اشتباه، باید این رشته را در پروسه ی هدف ایجاد کرد و آدرس رشته ی ایجاد شده را به CreateRemoteThread ارسال نمود.

ویندوز تابعی را در اختیار گذاشته است که می توان با استفاده از آن حافظه ای را در فضای آدرس پروسه ای دیگر اختصاص داد.

```
PVOID VirtualAllocEx(
   HANDLE hProcess,
   PVOID pvAddress,
   SIZE_T dwSize,
   DWORD flAllocationType,
   DWORD flProtect);
```

برای آزاد سازی فضای تخصیص داده شده:

```
BOOL VirtualFreeEx(
   HANDLE hProcess,
   PVOID pvAddress,
   SIZE_T dwSize,
   DWORD dwFreeType);
```

زمانی که حافظه تخصیص داده شد، به تسهیلاتی نیاز است که بتوان رشته ی مورد نظر را بر حافظه ایجاد شده رونوشت کرد.

ویندوز توابعی را در اختیار قرار می دهد که با استفاده از آن ها می توان حافظه ی پروسه ی دیگری را خواند و یا بر روی آن نوشت.

```
BOOL ReadProcessMemory(
   HANDLE hProcess,
   LPCVOID pvAddressRemote,
   PVOID pvBufferLocal,
   SIZE_T dwSize,
   SIZE_T* pdwNumBytesRead);

BOOL WriteProcessMemory(
   HANDLE hProcess,
   PVOID pvAddressRemote,
   LPCVOID pvBufferLocal,
   SIZE_T dwSize,
   SIZE_T* pdwNumBytesWritten);
```

پروسه ی هدف با آرگومان hProcess، حافظه ی ایجاد شده با pvAddressRemote و آدرس بافر محلی با pvBufferLocal مشخص می شوند. dwSize تعداد بایت ارسالی را مشخص می کند. pdwNumBytesRead و pdwNumBytesWritten تعداد بایت ارسال شده را مشخص می کنند.

بنابراین مراحل کار بدین صورت می باشد:

1. تخصیص حافظه در پروسه ی هدف با استفاده از VirtualAllocEx.
2. استفاده از WriteProcessMemory برای رونوشت گرفتن از رشته ی محلی حاوی مسیر کتابخانه در پروسه ی هدف.
3. استفاده از GetProcAddress برای بدست آوردن آدرس واقعی LoadLibraryA یا LoadLibraryW از کتابخانه ی Kernel32.dll.
4. استفاده از روال CreateRemoteThread برای ایجاد یک نخ دور در پروسه ی هدف که LoadLibrary (آدرس بدست آمده از مرحله 3) را با آرگومان مسیر کتابخانه (حافظه ی ایجاد شده در مرحله 1 و 2) اجرا می کند. در این نقطه کتابخانه به فضای آدرس پروسه ی هدف تزریق شده است و روال DllMain موجود در کتابخانه، اعلان DLL_PROCESS_ATTACH را دریافت می کند و می تواند روال دلخواه را اجرا کند. زمانی که روال DllMain به پایان می رسد، نخ نیز از فراخوانی LoadLibraryA/W، به روال BaseThreadStart بازمی گردد. سپس این روال ExitThread را فراخوانی می کند که باعث می شود نخ کشته شود.
5. استفاده از VirtualFreeEx برای آزادسازی حافظه تخصیص داده شده.
6. استفاده از GetProcAddress برای بدست آوردن آدرس واقعی روال FreeLibrary موجود در Kernel32.dll برای آزاد سازی کتابخانه تخصیص داده شده.
7. استفاده از CreateRemoteThread برای ایجاد نخ دور در پروسه ی هدف برای فراخوانی روال FreeLibrary با ارسال آرگومان HMODULE از کتابخانه ی دور.

انجام ندادن دو قدم آخر در پروسه ی ارزیابی، مشکلی ایجاد نمی کند.

API Hooking

برای در تله انداختن پروسه همانگونه که اشاره شد می توان از تزریق کتابخانه استفاده نمود.

هدف این کتابخانه آن است که توابعی که پروسه ی هدف استفاده می کند را اصطلاحا در تله انداخته و آن ها را به شیوه ی خود مدیریت کند.

دو راهبرد زیر برای این منظور می توان در نظر گرفت:

1. در تله انداختن با استفاده از بازنویسی کد: بدین صورت که آدرس تابع مورد نظر بدست آورده می شود، سپس می توان از دستورات مورد استفاده در اسمبلی و کدهای عملیاتی برای بازنویسی روال استفاده نمود. این روش همانگونه که قابل مشاهده است بسیار پیچیده و وقتگیر است.
2. در تله انداختن با دستکاری بخش واردات پروسه.

API Hooking by Manipulating a Module's Import Section

این تکنیک برای پیاده سازی و استفاده بسیار مناسب می باشد، اما نیازمند اطلاعات کافی در مورد الحاق پویا (Dynamic Linking) است. بخصوص باید دانست که در بخش واردات ماژول/پروسه چه چیزهایی موجود است.

بخش واردات یک ماژول شامل مجموعه ای از DLLهاییست که ماژول برای اجرا شدن نیاز دارد. علاوه بر آن حاوی لیستی از نشانه هاییست که ماژول از هرکدام از آن DLLها وارد می کند. زمانی که ماژول فراخوانی ای به روالی وارد شده را انجام می دهد، نخ آدرس آن روال را از بخش واردات می گیرد و به آن آدرس پرش می کند.

بنابراین برای در تله انداختن تابعی مشخص، باید آدرس آن تابع را در بخش واردات تغییر داد. از آن جایی که کدی بازنویسی نمی شود، نگرانی ای در باره ی همگام سازی نخ ها وجود ندارد.

تابع زیر ترفند مورد نظر را انجام می دهد، بخش واردات ماژول را جست و جو می کند تا نشانه مورد نظر و آدرس آن را پیدا کند. اگر چنین نشانه ای پیدا شد، آدرس نشانه را تغییر می دهد.

```
void CAPIHook::ReplaceIATEntryInOneMod(PCSTR pszCalleeModName,
   PROC pfnCurrent, PROC pfnNew, HMODULE hmodCaller) {

   // Get the address of the module's import section
   ULONG ulSize;

   // An exception was triggered by Explorer (when browsing the content of
   // a folder) into imagehlp.dll. It looks like one module was unloaded...
   // Maybe some threading problem: the list of modules from Toolhelp might
   // not be accurate if FreeLibrary is called during the enumeration.
   PIMAGE_IMPORT_DESCRIPTOR pImportDesc = NULL;
   __try {
      pImportDesc = (PIMAGE_IMPORT_DESCRIPTOR) ImageDirectoryEntryToData(
         hmodCaller, TRUE, IMAGE_DIRECTORY_ENTRY_IMPORT, &ulSize);
   }
   __except (InvalidReadExceptionFilter(GetExceptionInformation())) {
      // Nothing to do in here, thread continues to run normally
      // with NULL for pImportDesc
   }

   if (pImportDesc == NULL)
      return; // This module has no import section or is no longer loaded

   // Find the import descriptor containing references to callee's functions
   for (; pImportDesc->Name; pImportDesc++) {
      PSTR pszModName = (PSTR) ((PBYTE) hmodCaller + pImportDesc->Name);
      if (lstrcmpiA(pszModName, pszCalleeModName) == 0) {

         // Get caller's import address table (IAT) for the callee's functions
         PIMAGE_THUNK_DATA pThunk = (PIMAGE_THUNK_DATA)
            ((PBYTE) hmodCaller + pImportDesc->FirstThunk);

         // Replace current function address with new function address
         for (; pThunk->u1.Function; pThunk++) {

            // Get the address of the function address
            PROC* ppfn = (PROC*) &pThunk->u1.Function;

            // Is this the function we're looking for?
```

```
            BOOL bFound = (*ppfn == pfnCurrent);
            if (bFound) {
               if (!WriteProcessMemory(GetCurrentProcess(), ppfn, &pfnNew,
                     sizeof(pfnNew), NULL) && (ERROR_NOACCESS == GetLastError())) {
                  DWORD dwOldProtect;
                  if (VirtualProtect(ppfn, sizeof(pfnNew), PAGE_WRITECOPY,
                     &dwOldProtect)) {

                     WriteProcessMemory(GetCurrentProcess(), ppfn, &pfnNew,
                        sizeof(pfnNew), NULL);
                     VirtualProtect(ppfn, sizeof(pfnNew), dwOldProtect,
                        &dwOldProtect);
                  }
               }
               return; // We did it, get out
            }
         }
      } // Each import section is parsed until the right entry is found and patched
   }
}
```

برای تفهیم بهتر نحوه ی استفاده از این تابع، شایسته است مثالی آورده شود.

فرض شود ماژولی به نام Database.exe موجود است. کد در این ماژول، تابع ExitProcess را که در Kernel32.dll موجود است را فراخوانی می کند، اما مطلوب است بجای آن از MyExitProcess موجود در DbExtend.dll استفاده شود.

برای انجام این امر تابع ReplaceIATEntryInOneMod به این صورت فراخوانی می شود:

```
PROC pfnOrig = GetProcAddress(GetModuleHandle("Kernel32"),
   "ExitProcess");
HMODULE hmodCaller = GetModuleHandle("Database.exe");

ReplaceIATEntryInOneMod(
   "Kernel32.dll", // Module containing the function (ANSI)
   pfnOrig,        // Address of function in callee
   MyExitProcess,  // Address of new function to be called
   hmodCaller);    // Handle of module that should call the new function
```

اولین کاری که این تابع انجام می دهد آنست که، بخش واردات ماژول hmodCaller را با فراخوانی ImageDirectoryEntryToData و ارسال آرگومان IMAGE_DIRECTORY_ENTRY_IMPORT پیدا می کند و آدرس آن را که از نوع PIMAGE_IMPORT_DESCRIPTOR است را برمی گرداند، اگر این بخش پیدا نشد، NULL برگردانده می شود.
هم اکنون باید در این بخش تابعی که قرار است در تله بیافتد را جست و جو نمود. در این مثال نشانه هایی که از Kernel32.dll وارد شده اند مورد جست و جو هستند. حلقه for وظیفه جست و جو را انجام می دهد.
اگر در حلقه مرجعی به نشانه های وارد شده از Kernel32.dll یافت نشد، تابع پایان می پذیرد و کار دیگری انجام نمی دهد.
اگر مرجع مورد نظر پیدا شد، آدرسی به آرایه ی IMAGE_THUNK_DATA دریافت می شود که حاوی اطلاعاتی درباره ی نشانه های وارد شده می باشد.
سپس در این آرایه به جست و جوی آدرسی پرداخته می شود که همان آدرس تابع مورد نظر است (در این مثال، ExitProcess).
اگر هیچ آدرسی با آدرس مورد نظر برابر نبود، تابع پایان می پذیرد.
اگرآدرس پیدا شد، تابع WriteProcessMemory فراخوانی می شود تا آدرس تابع جایگزین را بنویسد.
اگر خطایی رخ دهد، حفاظ صفحه با استفاده از تابع VirtualProtect تغییر می یابد و بعد از جایگزینی به حالت اول برگردانده می شود.
از الان به بعد هر نخ در ماژول Database.exe اگر ExitProcess را فراخوانی کند، آن نخ تابع جایگزین را فراخوانی می کند.
از تابع جایگزین می توان به راحتی آدرس ExitProcess را از کتابخانه Kernel32.dll بدست آورد و بعد از انجام روال مورد نظر، آن را فراخوانی کرد.

اگر در ماژول Database.exe ماژول های دیگری هم وجود داشته باشد که ExitProcess را فراخوانی می کنند، آن ها نیز باید در تله قرار بگیرند. بنابراین برای هر ماژول باید ReplaceIATEntryInOneMod فراخوانی شود. برای این منظور تابع ReplaceIATEntryInAllMods با استفاده از توابع ToolHelp این مهم را انجام می دهد.

اگر یک ماژول با استفاده از LoadLibraryA/W ماژول دیگری را بارگزاری کرد، آن ماژول نیز باید در تله قرار گیرد، بنابراین باید توابع LoadLibraryA/W/ExA/ExW در تله قرار گیرند تا ماژول های جدید بارگزاری شده نیز با استفاده از ReplaceIATEntryInAllMods در تله قرار گیرند.

اگر یک نخ GetProcAddress را بدینصورت فراخوانی کرد:

```
typedef int (WINAPI *PFNEXITPROCESS)(UINT uExitCode);
PFNEXITPROCESS pfnExitProcess = (PFNEXITPROCESS) GetProcAddress(
   GetModuleHandle("Kernel32"), "ExitProcess");
pfnExitProcess(0);
```

این کد به سیستم می گوید که آدرس واقعی ExitProcess در Kernel32.dll را برگرداند. سپس آن را فراخوانی می کند. بدینصورت تابع در تله افتاده فراخوانی نمی شود. برای حل این مشکل، باید تابع GetProcAddress را نیز در تله انداخت تا آدرس تابع در تله افتاده را برگرداند.

یک نمونه کاربرد

برای پیاده سازی و نشان دادن تکنیک API Hooking برنامه ی کاربردی و کتابخانه ای طراحی شد که کلیات طراحی این برنامه، در قسمت های قبل قابل برداشت است.

در این برنامه تابع connect موجود در کتابخانه ws2_32.dll با تابع Hook_connect جایگزین می شود.

هدف تغییر مسیر اتصال پروسه ی هدف است.

سناریو بدین صورت است که 3 برنامه تحت شبکه با استفاده از زبان پایتون اجرا خواهند شد.

دو2 سرور وجود خواهد داشت که بر روی آدرس 127.0.0.1 یا 0.0.0.0 و بر پورت هایی دلخواه و مجزا، درحال پذیرش کلاینت هستند. سرور اول، سروری خواهد بود که کلاینت قصد ارتباط به آن را دارد و دیگری سروری است که باید با استفاده از API Hooking کلاینت به آن تغییر مسیر اجباری خواهد داد. کدهای کلاینت و سرور هنگام اجرای پایتون به صورت زنده وارد می شوند. یعنی از قبل فایلی حاوی کد پایتون موجود نخواهد بود.

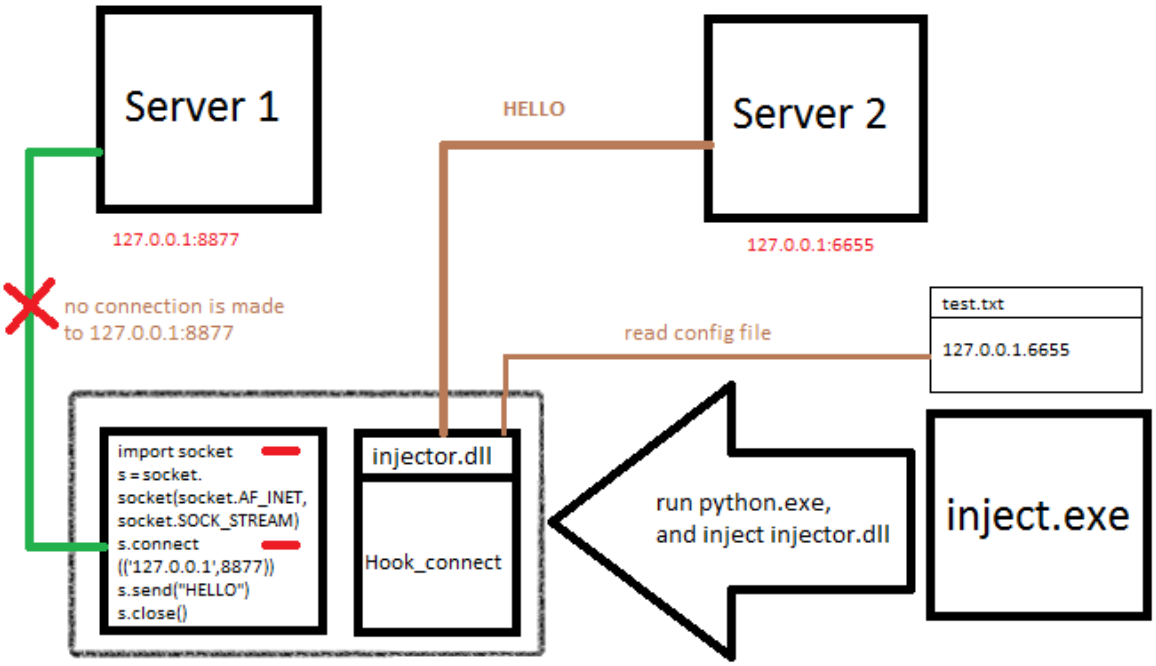
سرور اول : 0.0.0.0:8877

سرور دوم: 0.0.0.0:6655

اگر فرض شود که فایل پروسه ی هدف در مسیر C:\Python27\python.exe است، در آن پوشه فایل پیکربندی با نام test.txt باید قرار بگیرد که به عنوان مثال حاوی رشته 127.0.0.1.6655 می باشد. برنامه با استفاده از مسیر، پروسه را اجرا و کتابخانه را پس از 5 ثانیه (فرصت برای وارد کردن دستور اول برای بارگزاری ماژول socket در پایتون) در آن تزریق می کند.

حال، اگر پروسه ی هدف قصد دارد به 127.0.0.1:8877 متصل شود، تابع جایگزین با استفاده از آدرسی که از فایل پیکربندی خوانده آن را به 127.0.0.1:6655هدایت می کند.

این تغییر پارامترها برای لایه ی Transport شبکه و پروتوکل TCP انجام می شود.



کد سرور اول (مقصد اصلی):

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.bind( ( '0.0.0.0', 8877 ) )

s.listen(10)

c, a = s.accept() #wait for connections

#if had any connection

c.recv(1024) #receive the message

c.lose()

s.close()

exit()
```

کد سرور دوم(مقصد جعلی):

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.bind( ( '0.0.0.0', 6655 ) )

s.listen(10)

c, a = s.accept() #wait for connections

#if had any connection

c.recv(1024) #receive the message

c.lose()

s.close()

exit()
```

کد کلاینت:

```
import socket

#wait to receive injected message… then hit enter

>>INJECTED

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.connect( ( '0.0.0.0', 8877 ) )

s.send( "HELLO!!!" )

s.close()

exit()
```