White Hat Shellcode

The goal of this workshop is to plant a seed: that shellcode has a place in your defense toolbox. The goal is not to learn to write shellcode, neither is it to present a complete anthology of white hat shellcode. I want to show a few examples to help you be more creative, so that when you are facing a problem in your IT sec job, you will also consider shellcode as a potential solution.

Shellcode is almost always used in attack scenarios, but it can also be used to defend. Shellcode is just a tool, and it can be a solution to your problem.

In this workshop we will work together on 5 cases:

1. loading/unloading a DLL
2. enforcing DEP
3. testing your security setup
4. patching an application
5. preventing heapsprays with shellcode

Setting up the environment

I'm not providing a virtual machine with Windows installed because of licensing. You need to prepare your own machine.
Use a 32-bit Windows XP SP3 install, preferably a virtual machine, because we will install some software.
If you use an install with AV or other security software, be sure it permits the creation of remote threads.

Extract shellcode-workshop.zip to c:\.
Password is workshop.

Install the following software from directory c:\workshop\install (perform a default install):

- python-2.6.6.msi
- pywin32-216.win32-py2.6.exe
- nasm-2.07-installer.exe

Launch procexp.exe (display lower DLL panes, DEP status and base address) and dbgview.exe.

What is shellcode?

Shellcode is a program that is location independent and comes as a binary file without any metadata.

First example: loading/unloading a DLL

In our first example we will start without even using shellcode. We will just use code that is already present in the process we want to protect.

Injecting a DLL inside a process is a well known technique malware authors use to attack applications. The Windows API does not provide a function to load a DLL inside another process, so how is this done? There is an API function to load a DLL inside the current process: LoadLibrary. Loading a DLL inside another process is achieved by calling the LoadLibrary function inside that other process remotely. To execute code inside another process, the API function CreateRemoteThread is used. CreateRemoteThread will create a new thread (a unit of processing) inside the target process and execute the selected code with the provided argument. CreateRemoteThread can only pass one argument to the selected code, this is not a problem in our first example because LoadLibrary takes just one argument: the name of the DLL to load.

In this example, we will load DLL msgbox-hello.dll inside process calc.exe. For this, we need to create a string inside the calc.exe process (c:\workshop\ msgbox-hello.dll) and then call CreateRemoteThread to create a thread in calc.exe to execute LoadLibraryA with the string as argument.

- Start calc.exe
- Start procexp.exe (Process Explorer) and view the DLLs loaded in calc.exe
- Execute the following command from a command-line in c:\workshop: create-remote-thread.py calc.exe kernel32.dll LoadLibraryA str:c:\workshop\msgbox-hello.dll
- Click on the dialog box: the dialog box indicates that the DLL was successfully loaded.

You will notice a dialog box and the view in Process Explorer will display the DLL. This msgbox-hello.dll DLL is loaded at base address 0xBC0000.

So this is how attackers inject a (malicious) DLL inside your processes. But how do you unload this DLL? Again, the Windows API has no function to unload a DLL from another process. However, there is one for the current process: FreeLibrary. So we just need to call FreeLibrary via CreateRemoteThread with the correct argument (the base address of the DLL to unload).

- Execute the following command from a command-line in c:\workshop: create-remote-thread.py calc.exe kernel32.dll FreeLibrary 0xBC0000

You'll notice that msgbox-hello.dll disappears from Process Explorer's view: the DLL is unloaded.

So this is how you can undo a DLL injection. Without even using shellcode, just by remotely calling the appropriate API function with the right argument.

Now we will see how to achieve the same result with shellcode, just so you understand the difference.

- Reinject the DLL and take note of the base address
- Execute the following command from a command-line in c:\workshop: simple-shellcode-generator.py -o unload-dll.asm -l “kernel32.dll FreeLibrary 0xBC0000” (replace 0xBC0000 with the base address you wrote down)
- Start a NASM Shell from the Start \ All Programs \ Netwide Assembler menu
- From the NASM Shell: cd c:\workshop

- From the NASM Shell: nasm -o unload-dll.bin unload-dll.asm
- Execute the following command from a command-line in c:\workshop: create-remote-thread.py calc.exe unload-dll.bin

This last command writes the shellcode we generated (unload-dll.bin) in the calc.exe process, and then uses CreateRemoteThread to execute this shellcode. The shellcode unloads the DLL.

So the result is the same, but the technique is different. In stead of executing FreeLibrary directly with CreateRemoteThread, we generated shellcode that calls FreeLibrary and inject this shellcode remotely. Why would you use this extra step to include shellcode? Well, in this example, there is no reason. But we will soon see examples were shellcode is needed.

Second example: enforcing DEP

DEP is an important Windows security feature to protect against attacks. But not all applications use DEP, so we'll see now how to enforce DEP.

Let's start with a clean test application:

- exit calc.exe from the previous example, and start it again
- Notice DEP is enabled (DEP column in Process Explorer)
- Execute the following command from a command-line in c:\workshop: create-remote-thread.py calc.exe kernel32.dll SetProcessDEPPolicy 0
- refresh Process Explorer's view (F5) and notice that DEP has been turned off
- Execute the following command from a command-line in c:\workshop: create-remote-thread.py calc.exe kernel32.dll SetProcessDEPPolicy 1
- refresh Process Explorer's view (F5) and notice that Permanent DEP is enabled
- Execute the following command from a command-line in c:\workshop: create-remote-thread.py calc.exe kernel32.dll SetProcessDEPPolicy 0
- refresh Process Explorer's view (F5) and notice that Permanent DEP is still enabled

Once Permanent DEP has been enabled, it can not be disabled anymore. The only way to enforce Permanent DEP, is that the application (like calc.exe) calls SetProcessDEPPolicy with argument 1. So if you want to protect an application without DEP support with Permanent DEP, you must make that this application calls SetProcessDEPPolicy, and this is something you can do with shellcode.

First we generate shellcode to enforce permanent DEP:

- Execute the following command from a command-line in c:\workshop: simple-shellcode-generator.py -o dep.asm -l “kernel32.dll SetProcessDEPPolicy 1”
- From the NASM Shell: nasm -o dep.bin dep.asm

To make sure this shellcode works properly, inject shellcode dep.bin in calc.exe (restart calc.exe to reset default DEP).

Now, how can you modify calc.exe so that it calls SetProcessDEPPolicy each time it is launched? We can inject the shellcode dep.bin permanently in calc.exe with a PE-file editor, like LordPE.

- Execute the following command from a command-line in c:\workshop: copy \windows\system32\calc.exe calc-dep.exe
- Start LordPE.exe (LPE-DLX_1.4 folder): open calc-dep.exe
- Note the EntryPoint and the Image Base: 0x00012475 + 0x01000000 = 0x01012475
- Edit dep.asm, replace ret with these 2 lines
  mov eax, 0x01012475
  jmp eax

- From the NASM Shell: nasm -o dep.bin dep.asm
- From LordPE: add a section from file: dep.bin
- Notice the VOffset for dep.bin: 0x0001F000
- Change the EntryPoint to 0x0001F000
- From LordPE: Save
- From LordPE: Rebuild PE

Now you can run calc-dep.exe: notice that it has Permanent DEP now! We have added shellcode to calc-dep.exe that runs right before the original calc.exe program, this shellcode enforces DEP.

Third example: testing your security setup

People regularly ask me for malware so they can test their security setup. First, that's a bad idea, and second, you can do without.

Why is using malware a bad idea? It's dangerous and not reliable. Say you use a trojan to test your sandbox. You notice that your machine is not compromised. But is it because your sandbox contained the trojan, or because the trojan failed to execute properly? It might surprise you, but there's a lot of unreliable malware out in the wild.

So how can you reliably test your sandbox without risking infection, or even worse, have malware escape into your corporate network? Just use simple shellcode that creates a file in a location your sandbox should prevent, like system32.

Assemble and test sc-createfile.asm:

- From the NASM Shell: nasm -o sc-createfile.bin sc-createfile.asm
- Execute the following command from a command-line in c:\workshop: create-remote-thread.py calc.exe sc-createfile.bin
- Check if file c:\windows\system32\testfile.txt has been created: it has.

Now we will do the same, but with calc.exe running with restricted rights.

- First delete c:\windows\system32\testfile.txt
- Psexec -ld c:\windows\system32\calc.exe
- Execute the following command from a command-line in c:\workshop: create-remote-thread.py calc.exe sc-createfile.bin
- Check if file c:\windows\system32\testfile.txt has been created: it has not.

Fourth example: patching an application

Patches are changes to the binary code of an application. They typically fix bugs, security vulnerabilities or change features. But when you make changes to the files of an application (.EXE or .DLL), you invalidate the digital signature and you're probably breaking the EULA.
If you are in a position where you want to change an application, but are not in a position to change the binary files, shellcode designed to patch can help you.

2 years ago I developed a patch to fix an annoying “feature” of Adobe Reader 9.1. When you would disable JavaScript in Adobe Reader, each time you opened a PDF document with embedded JavaScript, Adobe Reader would remind you that JavaScript is disabled and propose you to turn in on again. To get rid of this nagscreen, I developed a patch: replace byte sequence 50A16CBF9323FF90C805000039750859 with 50A16CBF9323B80200000039750859 in file EScript.api. If you can not change file EScript.api, you can still change the code directly in memory.

- Install AdbeRdr910_en_US_Std.exe
- Open javascript.pdf, and notice the popup from the embedded JavaScript
- Disable JavaScript: Edit /Preferences / JavaScript / Enable Acrobat JavaScript
- Close and open javascript.pdf: notice the nagscreen from Adobe Reader
- From the NASM Shell: nasm -o sc-sar.bin sc-sar.asm
- Close javascript.pdf
- Start dbgview.exe
- Execute the following command from a command-line in c:\workshop: create-remote-thread.py calc.exe sc-sar.bin, and notice the message in dbgview after some time
- Open javascript.pdf: the nagscreen is gone.

Fifth example: preventing heapsprays with shellcode

Shellcode is often used in attacks and malware together with heapsprays: the heap is filled with shellcode (preceded by a long nopsled), and then the vulnerability is triggered. EIP jumps to the heap, hits a nopsled and slides to the shellcode. The shellcode executes, and typically downloads and installs a trojan.

Successful heapsprays can be prevented by pre-allocating memory. Then the heapspray can not write shellcode to the pre-allocated memory. But we can go one step further: if we pre-allocate memory and fill it with our own nopsled and shellcode, we can intercept the attack and block it. First we do an exercise to illustrate this idea.

- uninstall Adobe Reader 9.1
- install AdbeRdr812_en_US.exe
- start Adobe Reader
- unzip util-printf.zip (password is workshop)
- open util-printf.pdf and notice Adobe Reader crashing

This old version of Adobe Reader crashes when you open util-printf.pdf because this PDF document contains an exploit that makes EIP jump to 0x30303030 (might be a few bytes off). Since there is no code at this address, an exception is generated.

But when we put our own nosled and shellcode at this address, we achieve code execution.

- Execute the following command from a command-line in c:\workshop: simple-shellcode-generator.py -o sc-mba.asm -l "user32.dll MessageBoxA 0 str str 0"
- Edit sc-mba.asm with notepad and add 50 NOPs
- From the NASM Shell: nasm -o sc-mba.bin sc-mba.asm
- Start Adobe Reader 8
- Execute the following command from a command-line in c:\workshop: create-remote-thread.py -a 0x30303020 calc.exe sc-nopsled-mba.bin
- Notice the message box (and then click the message box away)

The message box appears because we injected our shellcode at address 0x30303020 and then executed it.

Now we try with the exploit PDF:

- Now open util-printf.pdf
- Notice the 2 message boxes (and then click the message boxes away)

This time, the message boxes appear because the exploit in PDF document util-printf.pdf transfer execution to address 0x30303030, where are shellcode is located. So the exploit executes our shellcode!

In this last example, we will use shellcode that suspends the attacked application and warns the user. This is something I've implemented in HeapLocker, a security tool to prevent heapsprays.

- Double click on heaplocker.reg and merge the entries to the registry

- Start Adobe Reader 8
- Execute the following command from a command-line in c:\workshop: create-remote-thread.py AcroRD32.exe kernel32.dll LoadLibraryA str:c:\workshop\heaplocker.dll
- Notice the messages in DbgView
- Now open util-printf.pdf
- Notice the warning
- Take a look at the threads with Process Explorer

You can do the same with a fully function exploit for util.printf: util-printf-heapspray.pdf. This PDF document contains an exploit (with heapspray) to launch calc.exe). Open this document with Adobe Reader, and then, do the same but first inject heaplocker.dll in Adobe Reader.