# Speak of the Devil: Runtime Disassembly and a Live Demo

scuti

2026/03/13

Back when I was using Windows 7, I was gamehacking with Cheat Engine.

I made a cheat table that wrote integers to `[400000+f59f00]+184` which was the address that stored game currency.

The Entertainment Software Association (ESA) (used to) target Cheat Engine and forums with copyright infringement claims.

Cheat Engine :: View topic - what happened
https://archive.is/suleX

# About the game

Devil May Cry 4 is a singleplayer action game written in C++ and released for Windows (x86).

- There is no in-game economy that ordinarily justifies real-money transactions.
- Offline except for Steam achievements.
- No sign of an interpreter such as Lua or Python.

# The weird notation I am using

Cheat Engine uses a notation for pointer paths that look like

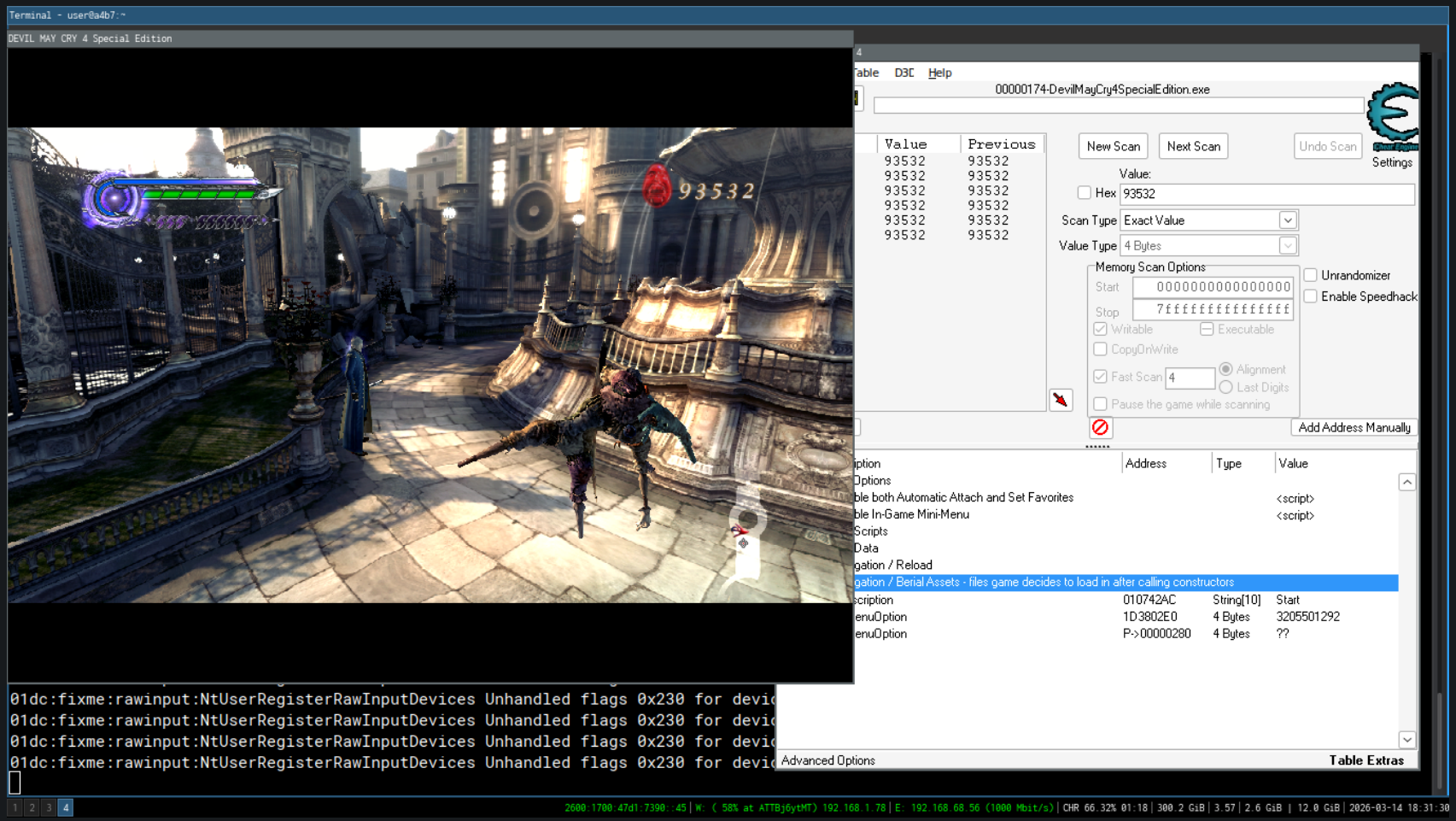
`[[0x400000 + base offset] + offset1] + offset2`

In this case `[400000+f59f00]+184` equals

```
; ecx is the game mediator
; uchar** base_address = (uchar**)0x1359F00;
; uchar* ecx = *base_address;

mov ecx, [400000+f59f00]

; ebx is the address of mediator -> currency
; int *ebx = ecx + 184;

lea ebx, [ecx+184]
```

# Gamehacking Basics

1. Scan memory for a value such as currency (Red Orbs, top right).
   - Check for increase when the player character collects.
   - Repeat until the results narrow down to one or a few results.



# Gamehacking Basics: Step Two


```
# watch for writes at (-l)ocation
(gdb) watch -l *0xADDRESS
```

```
rsi=0x1ae38d60
```

Dump of assembler code from 0x59a4f0 to 0x59a518:

```
   0x...59a4f0: add    DWORD PTR [rsi+0x184],edi
   0x...59a4f6: add    DWORD PTR [rsi+0x188],edi
   0x...59a4fc: cmp    DWORD PTR [rsi+0x184],0x98967f
   0x...59a506: jle    0x59a512
   0x...59a508: mov    DWORD PTR [rsi+0x184],0x98967f
   0x...59a512: cmp    BYTE PTR [rsi+0x1c],0x0
   0x...59a516: jne    0x59a521
```

End of assembler dump.

# Quick x86-64 register reference

| 64-bit | 32-bit | 16-bit | 8-bit | Purpose |
|---|---|---|---|---|
| rax | eax | ax | ah,al | General |
| rbx | ebx | bx | bh,bl | General |
| rcx | ecx | cx | ch,cl | General |
| rdx | edx | dx | dh,dl | General |
| rsi | esi | si | sil | “Source Index” |
| rdi | edi | di | dil | “Destination Index” |
| rbp | ebp | bp | bpl | Frame Pointer |
| rsp | esp | sp | spl | Stack Pointer |

# Memory Scanning Can Only Take You So Far

Memory scanning becomes less effective for booleans and floating point values between 0 and 1.

- If you're playing a game that needs 8 GB of RAM, the amount of bytes set to 0 at a given time will be in the hundreds of thousands at least.
- Scanning also stops being effective for variables that will only change once or a few times.
- Example: skipping a cutscene that will only play once in a mission.

# Mediator?

**Mediator** is a software design pattern where an object ("the mediator", typically a singleton) handles interactions between subcomponents and subordinate objects, typically to reduce dependencies and coupling.

- If a game uses this software design pattern and you find the address of the Mediator, setting a watchpoint will reveal instructions that handle collision, cutscene, event flags, etc.

# In C++ something may have looked like

```
class Mediator {
    // can't tell public or private from disassembler
    public:
        // ...
        GameEntity *camera;
        GameEntity *player_character;
        GameEntity *boss;
        List *enemies;
        // ...
        uint red_orbs;                  // +184
        uint red_orbs2;                 // +188
        uint red_orbs_collected;        // +18c
        uint red_orbs_spawned;          // +190
        // ...
}
```

# Writing to addresses

When you have pointer paths, you can modify values stored in addresses.

```
newmem:
; esi = mediator
originalcode:
; add edi = currency_spawned
; to mediator.total_currency_spawned
add [esi+00000190],edi

newcode:
push eax                ; save eax to stack
mov eax, [esi+190]      ; eax = ...total_currency_spawned
mov [esi+18c], eax      ; mediator.currency_collected = eax
pop eax                 ; restore eax from stack
```

# Writing New Instructions

But when you inject code, your new instruction must have the same number of bytes as the code you are replacing.

```
[disable]
DevilMayCry4SpecialEdition.exe+4CD933:
; FE 81 60 24 00 00 (little endian!)
inc byte ptr [ecx+00002460]

[enable]
DevilMayCry4SpecialEdition.exe+4CD933:
; do nothing instead of
; counting double jumps
db 90 90 90 90 90 90
```

# Code Injection continued

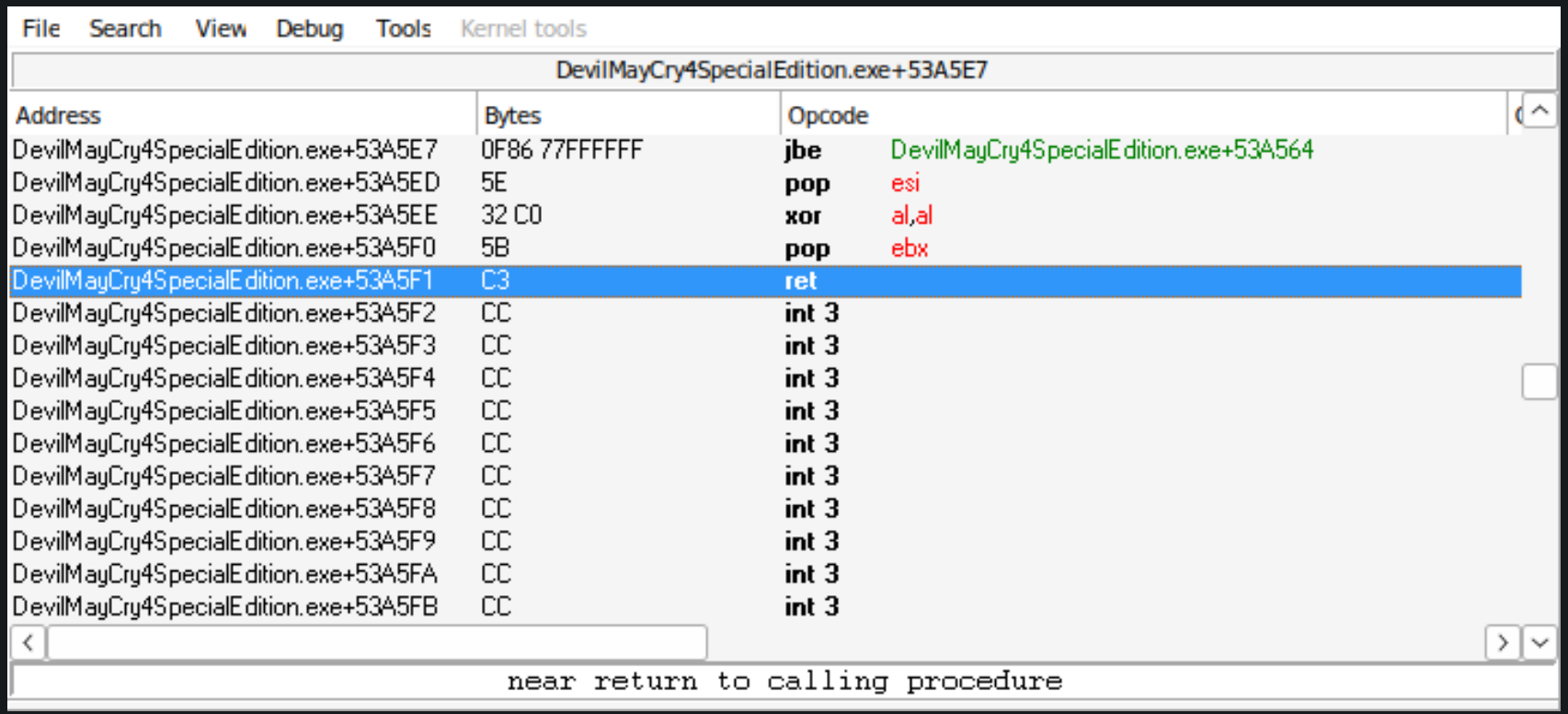
If the size of your new code exceeds the size of the original code, either use a **code cave** or allocate new memory.

```
[enable]
alloc(newmem,289)
registersymbol(next_stage)
alloc(next_stage, 2)

next_stage:
db 0 ; initialization, this will update per stage

originalcode:
mov [eax+00000088],edx

"DevilMayCry4SpecialEdition.exe"+152461:
jmp newmem
nop
returnhere:
```

# If you're wondering what a code cave is

Compilers typically add padding between functions such as `nop` (90) or `int3` (CC).

A jump to this unused space can be done instead of allocating new memory.



Any questions?

# Before moving onto the live demo

I am currently running Linux and playing the game through Wine.

Everything I wrote in x86 assembly works as it did before.

Some scripts I wrote in Lua do not work, especially D3Dhooks.

Attaching the Cheat Engine debugger within Wine causes a crash.

The CE-like frontend for gdb and libscanmem is kind of janky.

# Cheat Table Features

Basic cheats:

- Infinite double jumps, health, “mana”
- Disable timed events

The cool scripts:

- Auto-skip cutscenes
- Disable collision
- Kill everything in 1-hit
- Add boss rush mode

Unusual hacks:

- Rescale player or enemy models & hitboxes
- Instant enemy Devil Trigger
- Custom difficulty (more spawns + highest difficulty setting)

And more.