# Uvod

Mnoge igre v današnjem svetu se ne da več hekat z enostavnimi metodami kot včasih. Za to je predvsem kriva relativno nova iznajdba: kernel level anti-cheat. Ta deluje na več načinov. Najbolj tečna je to, da lahko prepreči, da "hook"-amo procese. Tu opisujem en način, kako bi se ga lahko izognil z pomočjo virtualizacije.

# 1. Preprosta aplikacija

Tu bom opisal kako narediti aplikacijo, ki jo je enostavno hekat in kaj gre lahko narobe. Končna aplikacija je napisana v Cju. Zgleda takole:

```c
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

volatile unsigned int a = 0;
volatile unsigned int b = 0;
volatile unsigned int c = 0;
volatile unsigned int d = 0;
volatile unsigned int e = 0;
volatile unsigned int f = 0;

int main() {
    while (1) {
        printf("Vals: %u %u %u %u %u %u \n", a, b, c, d, e, f);
        Sleep(2000);
        a++;
        b++;
        c++;
        d++;
        e++;
        f++;
    }
    return 0;
}
```

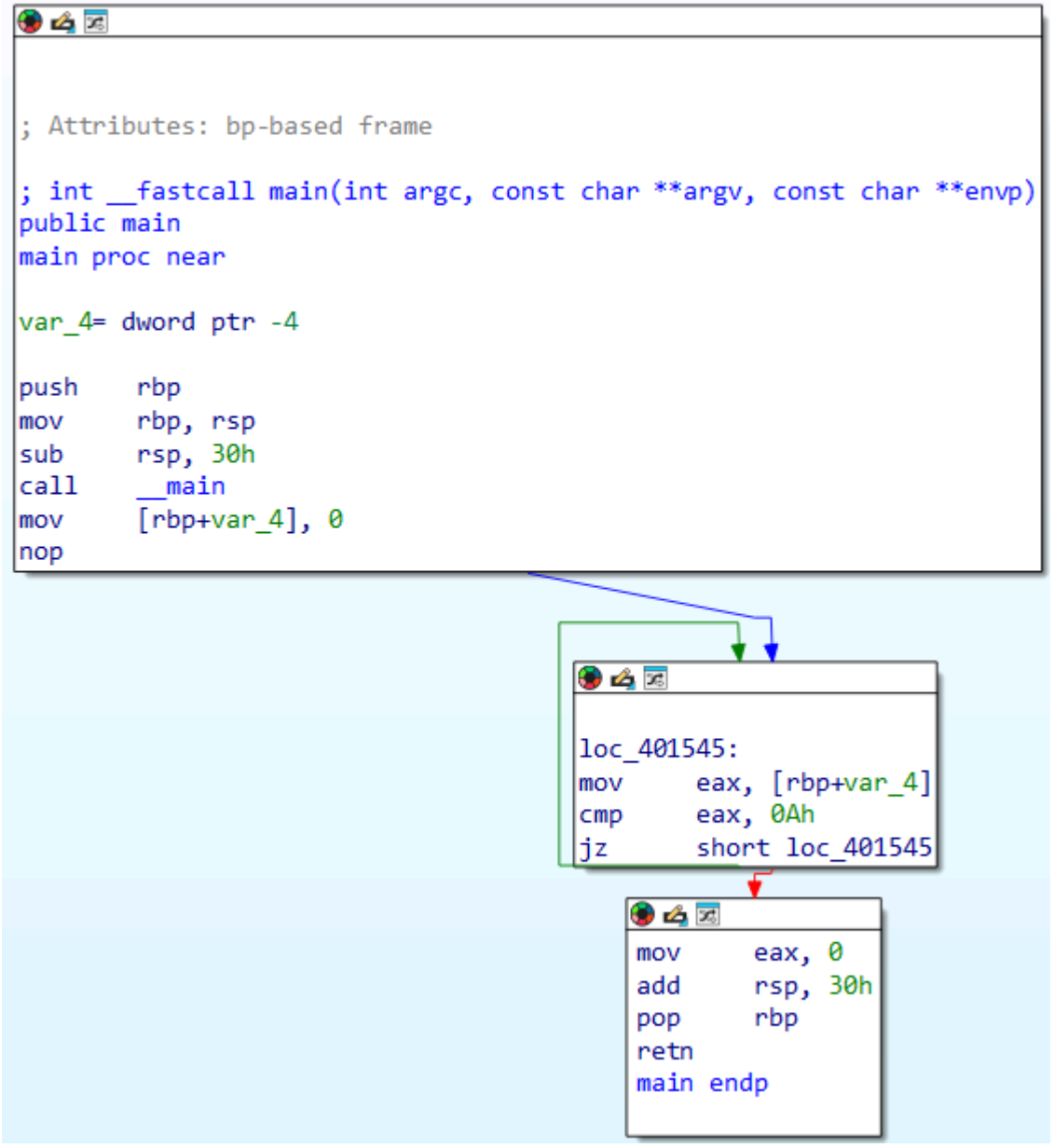
Vse kar ta aplikacija naredi je to, da vsake 2s poveča (a-f) za 1.

Pri izdelavi aplikacije je naš največji nasprotnik prevajalnik. Prevajalnik sam optimizira veliko kode. To ne gre čisto stran tudi, če prevajalnik zaženemo z optimizacijskim nivojem 0. Če pogledamo aplikacijo z katero sem začel:

```c
int main() {
    volatile int health = 0;
    while (health == 10) {}
```

```
    return 0;
}
```

Poglejmo še dekopalacijo (IDAfreeware):



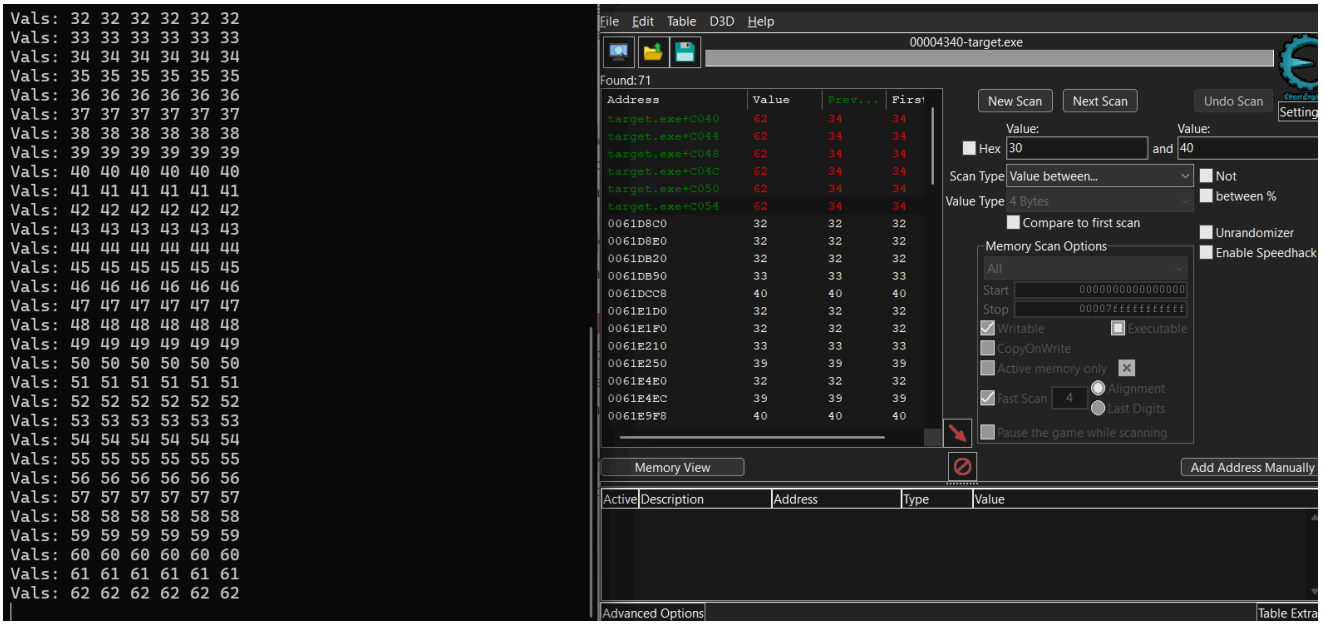

Spremenljivka 'health' se je preimenovala v 'var_4', ampak 'var_4' je skladovna spremenljivka. To lahko opazimo tam, kjer piše 'mov eax, [rbp+var_4]', kjer je 'rpb' frame pointer register. Realno v tako preprosti aplikaciji ne bi naredilo problema, ampak z vsak slučaj shranimo spremenljivko na statično mesto (vržemo jo izven funkcije). Naslednja stvar, ki jo naredimo je tudi to, da dodamo več spremenljivk. Za vsak slučaj, da se ne shranijo v registre (po vsej verjetnostni se ne bo zgodilo, ampak ne škodi).
Ne bom pokazal cele dekompalacije končnega programa, le del, ki bo prav prišel kasneje.

```
.bss:000000000040C03C                align 20h
.bss:000000000040C040                public a
.bss:000000000040C040 a              dd ?                    ; DATA XREF: main+6B↑r
.bss:000000000040C040                                        ; main+A0↑r ...
.bss:000000000040C044                public b
.bss:000000000040C044 b              dd ?                    ; DATA XREF: main+64↑r
.bss:000000000040C044                                        ; main+AF↑r ...
.bss:000000000040C048                public c
.bss:000000000040C048 c              dd ?                    ; DATA XREF: main+5D↑r
.bss:000000000040C048                                        ; main+BE↑r ...
.bss:000000000040C04C                public d
.bss:000000000040C04C d              dd ?                    ; DATA XREF: main+57↑r
.bss:000000000040C04C                                        ; main+CD↑r ...
.bss:000000000040C050                public e
.bss:000000000040C050 e              dd ?                    ; DATA XREF: main+51↑r
.bss:000000000040C050                                        ; main+DC↑r ...
.bss:000000000040C054                public f
.bss:000000000040C054 f              dd ?                    ; DATA XREF: main+4A↑r
```

Tu usmerim vid na 2 stvari. Prva je to, da so podatki v .bss delu (med statičnimi podatki, ne na skladu), kot smo si želeli in druga stvar na katero smo lahko pozorni so offseti od 0x400000. Čez te bomo bližje pogledali pri samem hekanju.

# 2. Hekanje aplikacije

Aplikacijo bomo najprej pogledali z Cheat engine. Z tem programom je možno gledat spomin aplikacije, filtrirat vrednosti na teh naslovih... Z tem programom so dinozavri hekali CS 1.6.




Preprost tutorial za Cheat engine:

1. Izberi kater proces hočeš heckat (bodi pozoren, da bo pravi!)
2. Desno napišeš kako hočeš skenirat spomin
3. Skeniraš dokler ne najdeš iskanih naslovov
4. Spremeniš vrednosti na tistih naslovih

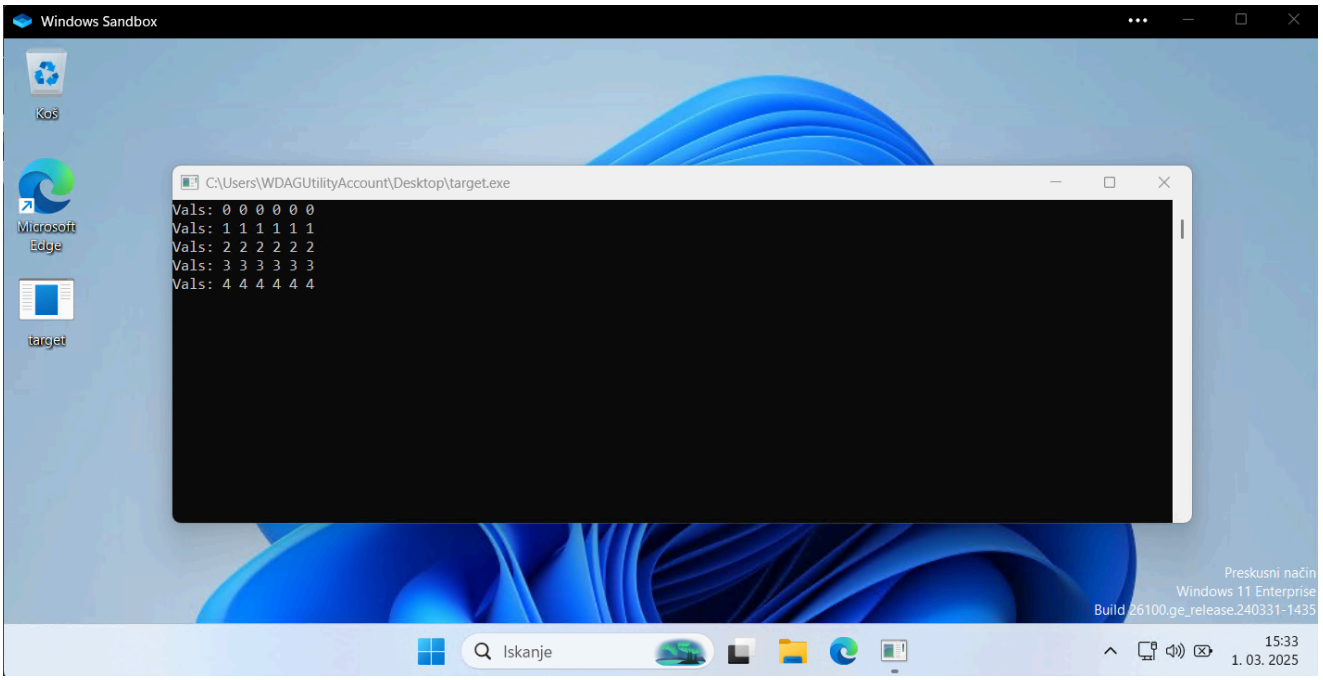
Na sliki vidimo 6 naslovov. Te se imenujejo 'target.exe+offset'. Te offseti so enaki, kot so bili ko smo dekompalirali aplikacijo. Naslov 'target.exe' se zmeraj spremeni, ko zaženemo aplikacijo. Cheat engine je super program za raziskovanje programa, vendar če hočemo naredit dober cheat ga napišemo (kopiramo) sami:

```
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;
// (Ne preveč razmišljat
const int PROCESS_ALL_ACCESS = 0x1F0FFF; // Read+Write po naslovih
[DllImport("kernel32.dll")]
static extern IntPtr OpenProcess(int dwDesiredAccess, bool bInheritHandle,
int dwProcessId);

[DllImport("kernel32.dll", SetLastError = true)]
static extern bool WriteProcessMemory(int hProcess, int lpBaseAddress,
byte[] lpBuffer, int dwSize, ref int lpNumberOfBytesWritten);
// o tem delu kode)

Process[] proceses = Process.GetProcessesByName("target");
if (proceses.Length == 0)
{
    Console.WriteLine("Ne najdem procesa");
    return;
}

IntPtr processHandle = OpenProcess(PROCESS_ALL_ACCESS, false,
proceses[0].Id);
Console.WriteLine("Process handle: " + processHandle);

IntPtr baseAddress = proceses[0].MainModule.BaseAddress;
Console.WriteLine("Process base address: " + baseAddress);

int bytesWritten = 0;
byte[] buffer = [150, 0, 0, 0]; // Zapiše 150
int offset = 0xC040; // Offset od spremenljivke 'a'

WriteProcessMemory((int)processHandle, (int)IntPtr.Add(baseAddress, offset),
buffer, sizeof(uint), ref bytesWritten);
```

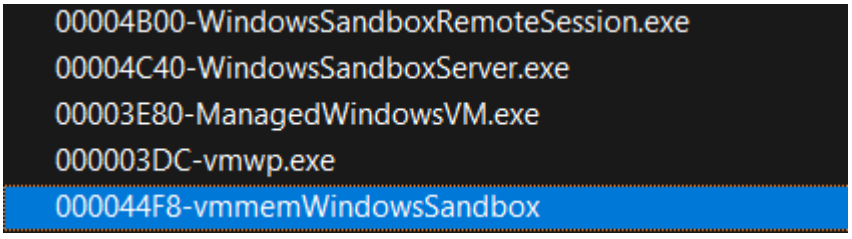
Koda je napisana v C# in ne C/C++ ker se imam rad. Na kratko: Najprej najdemo vse procese, ki se imenujejo 'target'. To je ime naše preproste aplikacije (pogledaš v task manager). Nato izpišemo njegov handle. To je unikatna številka, ki jo OS uporablja, ko hoče upravljat proces. Uporabimo jo, da najdemo 'baseAddress', ki je virtualni naslov na katerem se začne naš program (pri Cheat engine se je imenoval preprosto 'target.exe'). Nato zapišemo 150 tam, kjer se nahaja spremenljivka 'a'.

# 3. Hekanje aplikacije v WS

Windows Sandbox je zakon aplikacija, ki omogoča preprosto virtualizacijo brez, da bi mogli zmeraj naložit nov operacijski sistem. Našo aplikacijo lahko preprosto kopiramo na namizje.




Super program dela. Gremo ga probat hekat. Samo, da ga odprem z Cheat engine...

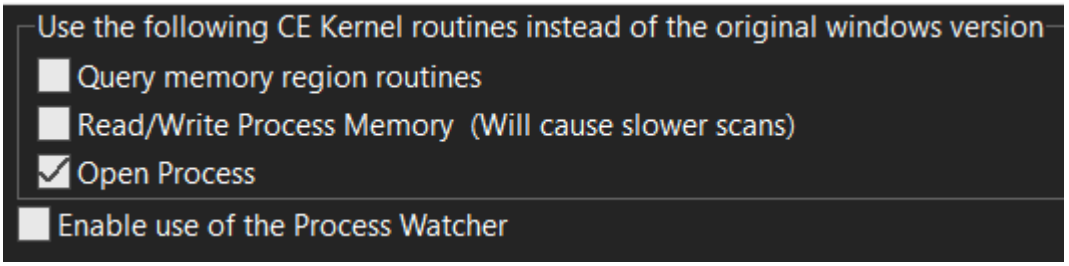



Ok. Tu smo dobili veliko informacij:

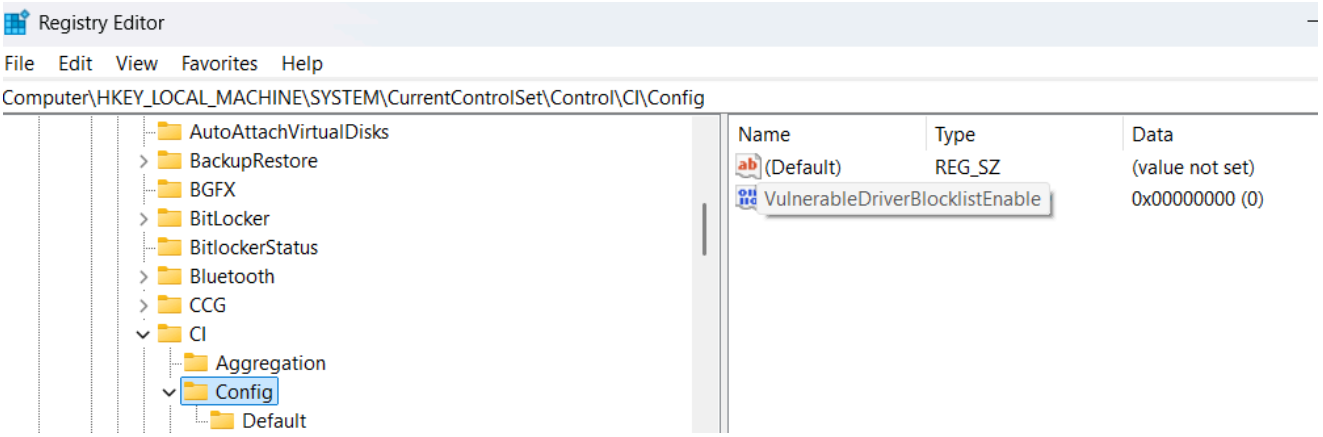
1. Ne moremo direktno odpret procesa naše aplikacije, ker se skriva v procesu od WS
2. WS ima več kot 1 proces
   To skrivnost, da ima lahko aplikacija več procesov sem skrival že od takrat, ko sem zagnal aplikacijo brez WS. Naša aplikacija je en proces, ki se kaže v Windows Terminalu, ki je drug. In samo terminal se pokaže kot aplikacija zato je bolj viden, ampak spremenljivke, ki jih hočemo so v drugem procesu. Tu imamo 4-5 procesov.
   Sklepanje:

- WindowsSandboxRemoteSession: po vsej verjetnosti nek most med serverjem in UI
- WindowsSandboxServer: Tu bi se mogoče lahko kaj uporabnega skrivalo
- ManagedWindowsVM: To se sliši kot nekaj kar bi bil lahko nek API za samo VM
- vmwp: idk. Tisti za katerega najmanj veš je ponavadi prav
- vmmemWindowsSandbox: VM memory? Sliši se pomembno
  Testiral sem seveda vse in zadnji je bil pravi. 'vmmemWindowsSandbox' hrani skrbi za to, kaj se dogaja v VM. Razlog zakaj sem se ga spravil pogledat nazadnje je pa preprost. Nisem ga mogel odpret

Cheat engine se že tako izvaja v admin načinu, zato to ni problem. Če malo premislimo. Virtualizacija na Windows uporablja Hyper-V hypervisor, ki po vsej verjetnosti dela v kernel mode. Cheat engine dela v user mode. Torej bi rabil Cheat engine povzdignit na kernel mode. Na srečo Cheat engine to omogoča.




Pomemben je samo 'Open Process', ampak preden ga izberemo, rabimo še malo popravit register.




Sedaj lahko beremo spomin WS, ampak smo spet prišli, do problema. Prej smo imeli offsete, ki se začnejo od baznega naslova. Bazni naslov je bilo trivialno najti (prebrali smo ga iz OS s pomočjo proces handla). Tu tega ne moremo naredit, ker nimamo handla procesa imamo samo handle WS. Zato sem popravil našo aplikacijo, da ima poleg drugih naslovov še lokator.

Recimo, da izdelujemo neko aplikacijo. Če dodamo neko novo funkcijo, bo premaknila vso drugo kodo in pokvari tvoj cheat, tudi če sploh ne veš, da je kdo probal vdreti v aplikacijo. Zato je treba zmeraj popravit vse offsete. To naredijo programi, ki se imenujejo 'offset dumper'. Te gledajo za zanimiva zaporedja bitov okoli iskane spremenljivke. Ko najde to zaporedje ve, da se po vsej verjetnosti nekje tam skriva podatek. Če vzamemo dovolj bitov je velika možnost, da se tisto zaporedje ne ponovi nikjer drugje v kodi. (ta paragraf je skriva veliko stvari/veliko jih je zelo narobe. Pomembna stvar je ideja, ki jo predstavlja, ne točno kaj se dogaja)

Tako dela naš lokator. Lokator ima v sebi zapisano vrednost, ki se redko pojavi. Lokacijo te vrednosti tako najdemo brez problema.

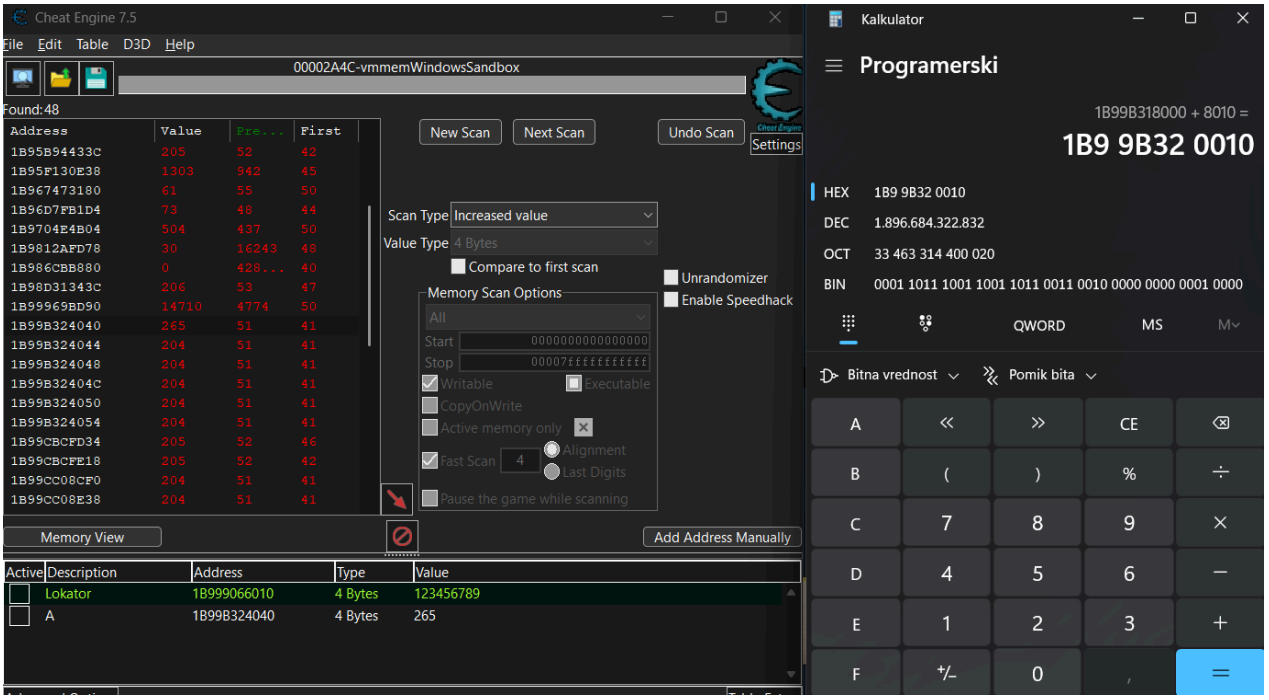
```
volatile unsigned int lokator = 123456789; // base+8010

volatile unsigned int a = 0; // base+C040
volatile unsigned int b = 0; // base+C044
volatile unsigned int c = 0; // base+C048
volatile unsigned int d = 0; // base+C04C
volatile unsigned int e = 0; // base+C050
volatile unsigned int f = 0; // base+C054
```

Še odpremo program v IDA:

```
.data:0000000000408004                  align 10h
.data:0000000000408010                  public lokator
.data:0000000000408010 lokator          dd 75BCD14h             ; DATA XREF: main+D↑r
.data:0000000000408010                                          ; main+16↑w
```

Tu vidimo, da se lokator nahaja v .data in da je njegov offset 0x8010. Da dobimo neko drugo spremenljivko rešimo to enačbo (vaja za bralca):

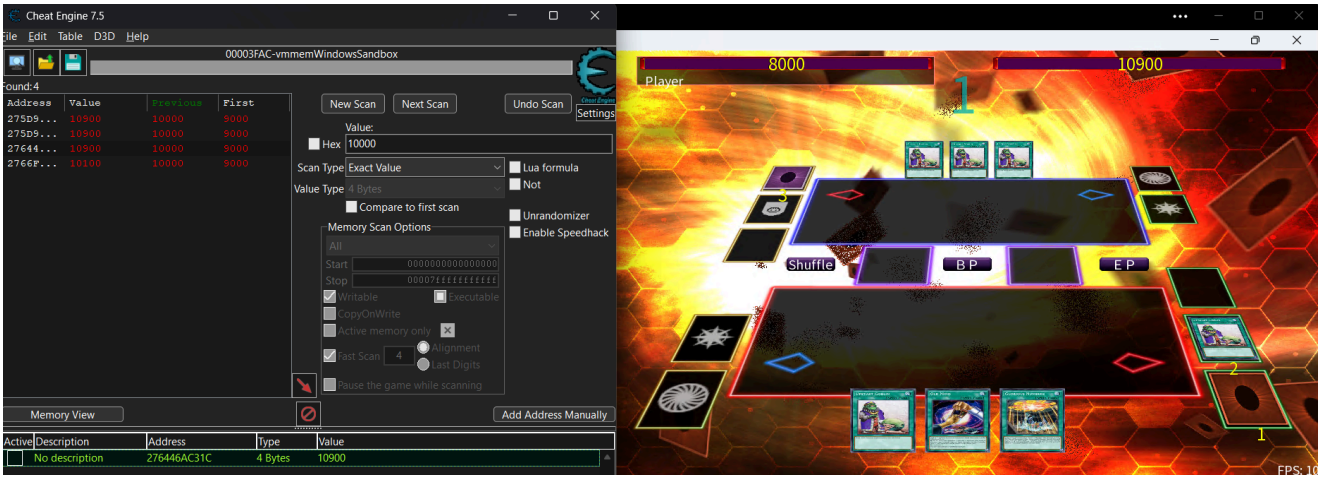
- Vemo: offset od a, offset od lokatorja, naslov lokatorja
- Enačba: naslov spremenljikve = bazni naslov + offset




(Slika predstavlja ravno obratno. Ker je bilo več (3) naslovov z vrednostjo 123456789 :P )

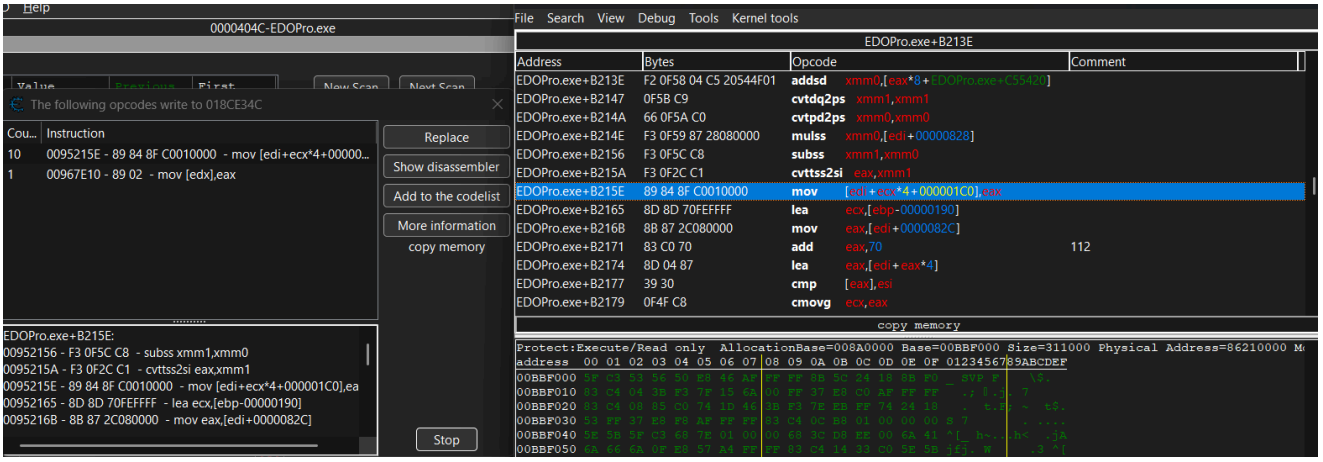
# 4. Praktični primer

Za praktični primer sem izbral simulator Project Ignis. Načeloma je lažje hekat aplikacije, kjer vidiš točna števila. Spodaj je primer, kako lahko lahko spreminjaš vrednosti življenskih točk, ko aplikacija deluje v WS. Postopek je podoben tistemu, ki je bil pokazan zgoraj.

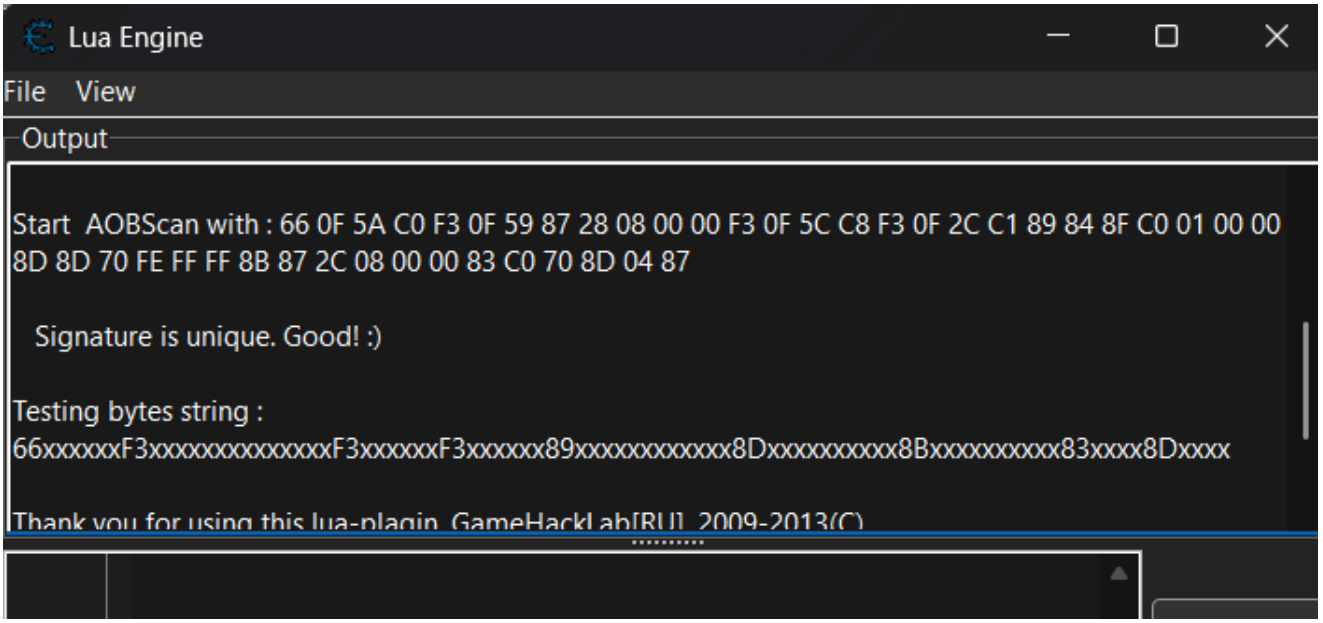



Bolj zanimivo je, če preprosto onemogočimo spreminjanje življenskih točk. To se lahko preslika v drugih igrah v npr. onemogočanje kolizije, onemogočanje gravitacije,... Lažje je nekaj uničit kot dodat.

To se preprosto naredi z tem, da pogledaš kaj piše na tisti naslov in malo "popraviš" kodo.




V Cheat engine je to narediš z tem, da proces priklopiš na debuger in gledaš, kaj piše na tist naslov. Nato pa preprosto spremeniš 'mov' operand z "nop". Ko pridemo do tu, lahko še naredimo signaturo.




Signatura skupaj z masko (testing string) nam pomaga locirat ta naslov, tudi če se aplikacija posodobi (note: v kodi uporabljam drugo signituro). Z uporabo signitur lahko najdemo naslov brez, da bi uporabili Cheat Engine. V ta namen naredimo DLL, ki ga vrinemo v proces. Dela v več stopnjah. Prvo poiščemo signituro (in z tem naslov na katerem so ukazi, ki jih hočemo popravit). Za ta namen uporabimo knjižnjico ali pa kopiramo kodo nekoga pametnejšega, vsaj se ne bo spreminjala med cheati in se lahko zelo optimizira.
Ko imamo naslov na katerem hočemo popravit kodo...jo popravimo:

```
void replace(uintptr_t addr) {
    returnAddr = addr;

    DWORD stariPrem;
    VirtualProtect((LPVOID)addr, 6, PAGE_EXECUTE_READWRITE, &stariPrem);
    DWORD relativeAddr = ((DWORD)replacementFunction) - addr - 5;
    ((byte*)addr)[0] = 0xE9; // JMP ukaz
    *(DWORD*)(addr + 1) = relativeAddr;
    ((byte*)addr)[5] = 0x90; // NOP ukaz po JMP, ker je MOV dalši od JMP
(god bless x86)
    VirtualProtect((LPVOID)addr, 6, stariPrem, &stariPrem);
```

```
    DWORD temp;
    VirtualProtect((LPVOID)replacementFunction, 300, PAGE_EXECUTE_READWRITE,
&temp);
}
```

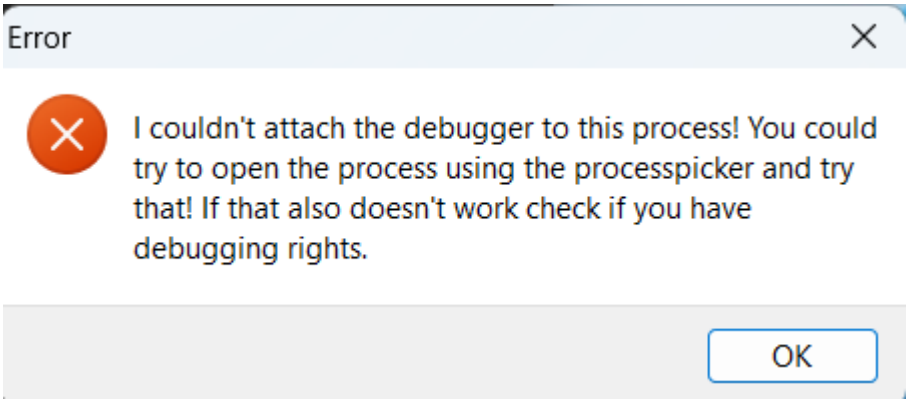
Tu sem preprosto spremenil MOV operand v JMP + NOP (zaradi velikosti). Ena od nalog OS je, da skrbi pred nepooblaščenim dostopom, ampak ti pusti vse, če ga lepo vprašaš (razlog za VirtualProtect funkcijo). Sama funkcija v katero skočimo zgleda takole:

```
uintptr_t returnAddr;
__declspec(naked) void replacementFunction() { // <-Pozor! naked dela samo
na x86 in arm
    cout << "Zamenjaj ukaz je bil poklican" << endl;
    __asm {
        jmp returnAddr+7 // +7, da preskoči jmp ukaz
    }
}
```

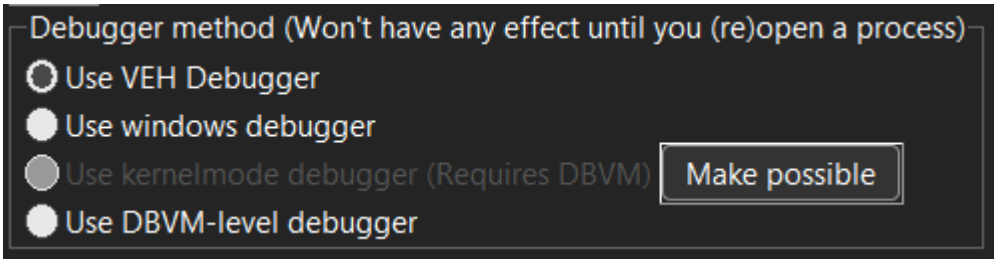
Spet se borimo proti prevajalniku. C++ prevajalnik pred in po funkciji rad doda dodatno kodo (pripravi stack, registre,...). '__declspec(naked)' to prepreči. Sama funkcija ne naredi nič, samo izpiše, da se je izvedla (MOV ukaz smo tako ali pa tako izbrisali).

Rad bi še omenil, da se po navadi to ne dela tako kot sem jaz. Po navadi se hook-a funkcije ne pa naključne ukaze sredi njih. Na tak način delajo razni overlayi (npr. za Discord).

Ker bi rabil čakat 2 uri, da se mi EdoPro naloži v Windows Sandbox nisem testiral, če dela v sandbox. Edin razlog zakaj ne bi delalo, bi bil, da bi se signatura nekako spremenila. Gremo jo pogledat z CheatEngine...




Debuger je user mode, aplikacija je kernel mode (mogoče)... Kar rabimo je kernel mode debuger. Gremo v nastavitve...




Dve stvari. Prva je, da je treba popravit register. Druga je pa DBVM. Samo Intel procesorji

podpirajo DBVM. Jaz imam AMD... Unlucky. Nadaljujemo z predvidevanjem, da je po vsej verjetnosti mogoče.

# 4. Jedrska aplikacija v WS

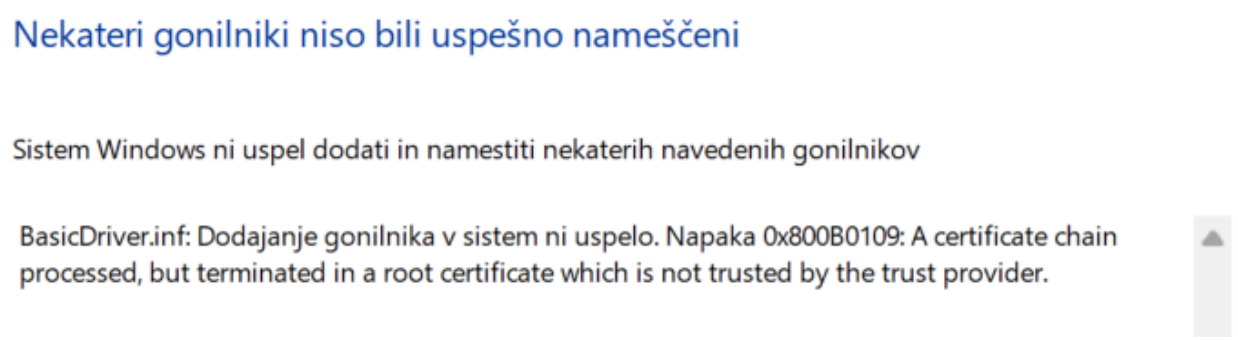
Sedaj smo rešili problem dostopa do aplikacije, ki jo straži antivirus, ampak antivirus lahko še zmeraj skenira aplikacijo za spremenjeno kodo. Kaj pa, če bi probali onemogočit antivirus? V tem delu bomo namesto, da bi spreminjali aplikacijo v uporabniškem načinu spreminjamo aplikacijo v jedru. Takim aplikacijam rečemo gonilniki.

```
#include <ntddk.h>
#define DRIVER_TAG 'DTag'

volatile unsigned int health = 12345678;
volatile unsigned int a = 0;
volatile unsigned int b = 0;
volatile unsigned int c = 0;
volatile unsigned int d = 0;
volatile unsigned int e = 0;
volatile unsigned int f = 0;

BOOLEAN StopThread = FALSE;
HANDLE hThread;
PVOID pThreadObject = NULL;

// Worker thread
VOID WorkerThread(PVOID Context) {
    UNREFERENCED_PARAMETER(Context);

    LARGE_INTEGER interval;
    interval.QuadPart = -20000000; // 2 sekunde

    while (!StopThread) {
        KeDelayExecutionThread(KernelMode, FALSE, &interval);

        a++;
        b++;
        c++;
        d++;
        e++;
        f++;

        DbgPrint("Health: %u\n", health);
    }

    PsTerminateSystemThread(STATUS_SUCCESS);
}

// Driver unload function
VOID DriverUnload(PDRIVER_OBJECT DriverObject) {
    UNREFERENCED_PARAMETER(DriverObject);

    StopThread = TRUE;
    if (pThreadObject) {
```

```
        KeWaitForSingleObject(pThreadObject, Executive, KernelMode, FALSE,
NULL);
        ObDereferenceObject(pThreadObject);
        ZwClose(hThread);
    }

    DbgPrint("Driver unloaded.\n");
}

// DriverEntry function
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING
RegistryPath) {
    UNREFERENCED_PARAMETER(RegistryPath);
    NTSTATUS status;

    DriverObject->DriverUnload = DriverUnload;

    status = PsCreateSystemThread(&hThread, THREAD_ALL_ACCESS, NULL, NULL,
NULL, WorkerThread, NULL);
    if (!NT_SUCCESS(status)) {
        DbgPrint("Failed to create worker thread!\n");
        return status;
    }

    status = ObReferenceObjectByHandle(hThread, THREAD_ALL_ACCESS, NULL,
KernelMode, &pThreadObject, NULL);
    if (!NT_SUCCESS(status)) {
        ZwClose(hThread);
        return status;
    }

    DbgPrint("Driver loaded successfully.\n");
    return STATUS_SUCCESS;
}
```

To je začetna aplikacija prepisana kot KMDF gonilnik. Tu se moje znanje malo ustavi zato se ne bom spuščal v detajle (ChatGPT my beloved). Glavna stvar, ki je za vedet je to, da ko se driver naloži naredi nov thread, ki šteje, kot je prej.

Gremo ga probat naložit v WS.

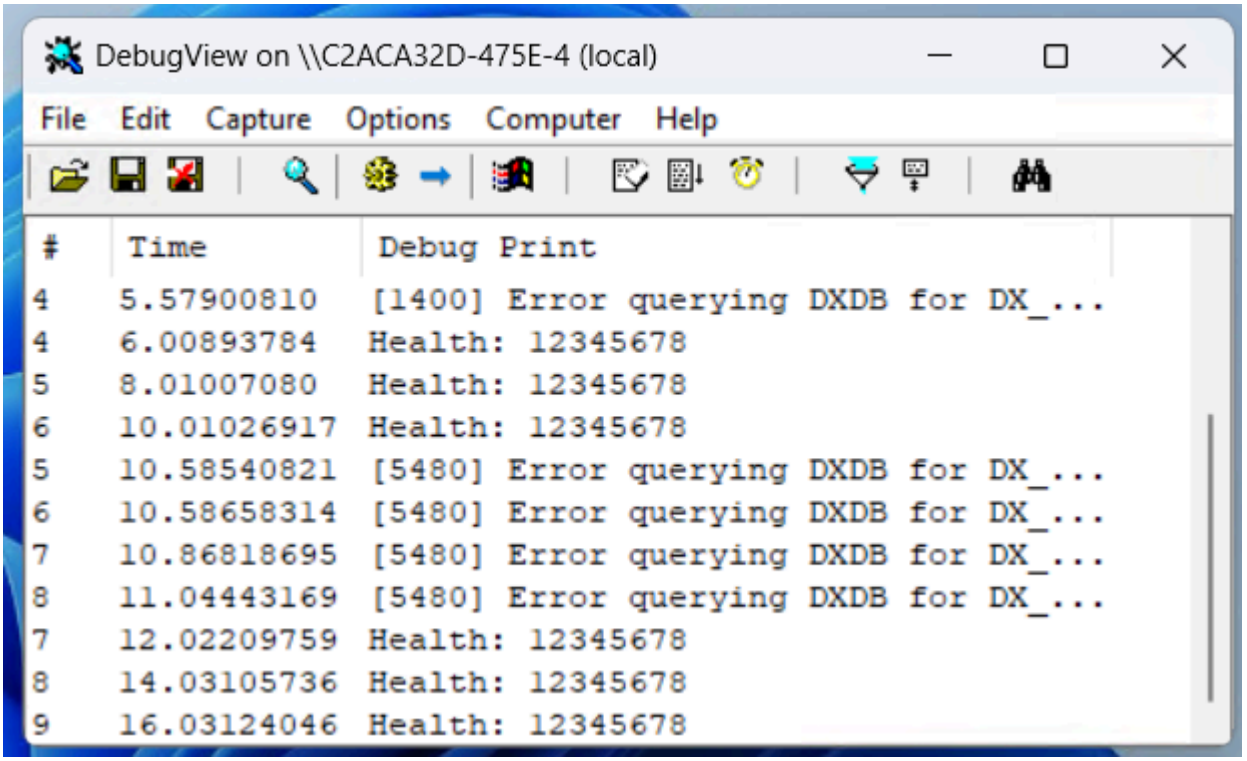



Tečno. Gremo zrihtat. Rešitev je dve komandi v powershell, ki omogočijo test sign driverjev v WS. Ta komanda je kakor vem nedokumentirana iz strani Microsofta. Zdi se mi, da je bil prvi, ki jo je našel nekdo na twiterju: https://x.com/jonasLyk/status/1366700591876079623?t=ocIuZGY5ZwjgRvFAiKw5VQ.

Tako zgleda, ko je uspešno zagnan.

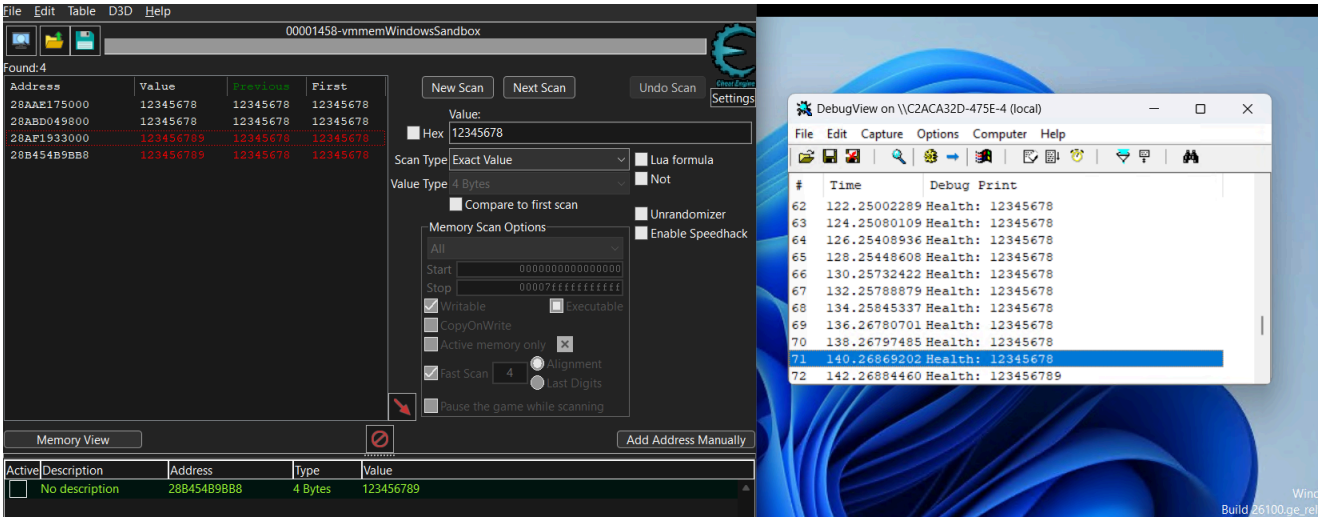
```
C:\Users\WDAGUtilityAccount>sc create MyDriver type=kernel start=demand binPath="C:\BasicDriver\BasicDriver.sys"
[SC] CreateService SUCCESS

C:\Users\WDAGUtilityAccount>sc start MyDriver

SERVICE_NAME: MyDriver
        TYPE               : 1  KERNEL_DRIVER
        STATE              : 4  RUNNING
                                (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE    : 0  (0x0)
        SERVICE_EXIT_CODE  : 0  (0x0)
        CHECKPOINT         : 0x0
        WAIT_HINT          : 0x0
        PID                : 0
        FLAGS              :
```

Še naložimo DbgView Microsoftove spletne strani in vključimo opcijo 'capture kernel'.




# 5. Hekanje jederske aplikacije v WS

Nepresenetljivo, hekanje kernel aplikacije je identično kot hekanje navadne aplikacije. Plan je bil ugasniti izpis te aplikacije, ampak mi ni uspelo ker imam AMD procesor.




Glavna stvar, ki sem jo hotel dokazat z tem delom je, da je teoretično mogoče uničit

anticheat. Ena stvar, ki se jo splača preverit je to, če WS uporablja virtualne naslove (in jih) in če so zaradi tega programi razkosani na 4kB velike kose. Teoretično z skeniranjem signatur ne bi smelo predstavljat večjih problemov pri izdelavi cheatov, ampak se splača preverit.

# Uporabni viri


Coding convention za C++:
https://learn.microsoft.com/en-us/windows/win32/LearnWin32/windows-coding-conventions

AA maker plugin za izdelavo signatur:
https://forum.cheatengine.org/viewtopic.php?t=587401

Driver del:
https://learn.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/writing-a-very-small-kmdf--driver
https://secret.club/2022/08/29/bootkitting-windows-sandbox.html#windows-sandbox-for-driver-development
https://learn.microsoft.com/en-us/sysinternals/downloads/debugview