Using VBS enclaves for anti-cheat purposes

tulach.cc/using-vbs-enclaves-for-anti-cheat-purposes

Samuel Tulach November 9, 2024

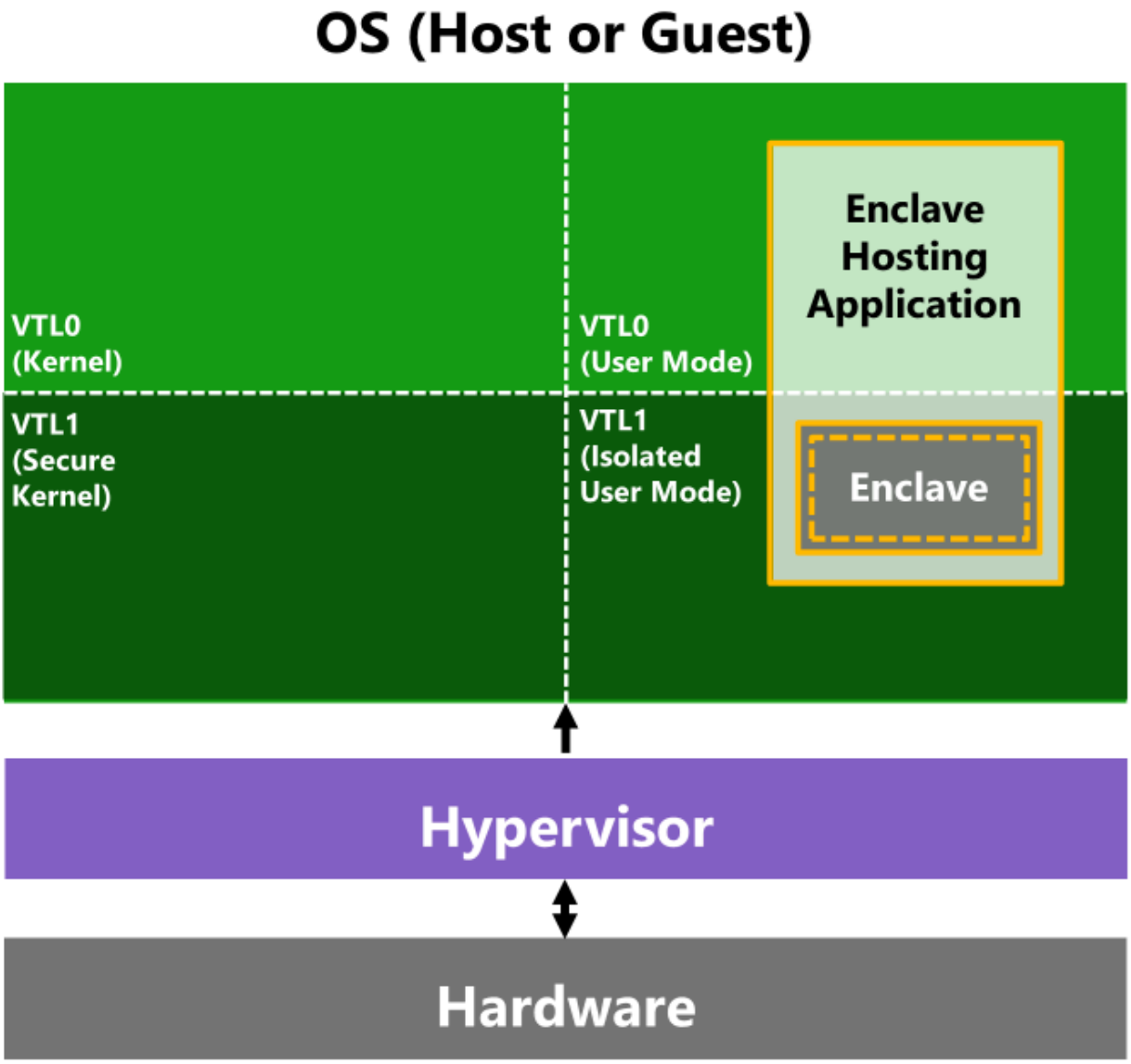
A few months ago, when Microsoft announced VBS (virtualization-based security) enclave functionality, I started wondering whether it could be used for game anti-cheating purposes. While I was skeptical (later I will explain why), I decided to look into it and write a simple Pong-inspired game which handles its entire game logic in such enclave.

What are VBS enclaves?

Hyper-V is a type-1 hypervisor, which means that when enabled, the Windows installation it's running on becomes a guest.

Virtualization-based security (VBS) is built on top of the Hyper-V platform. Since it operates at a higher privilege level than the kernel itself, it can help enforce strict code signing requirements (HVCI) or completely isolate data even from code running at the kernel level.

VBS enclaves allow developers to leverage VBS in their applications to isolate code and data from anything else running on the computer. Think of it as running Windows inside VirtualBox or VMware Player as one VM, with this isolated environment as another. Neither can access the other unless the hosting application explicitly permits it.



Trusted execution enclaves using VBS (source)

Anti-cheat?

Traditionally on Windows, anti-cheat software utilizes kernel-mode drivers to (among other things) prevent any user-mode programs from accessing the game's memory. I have written about it in more detail here.

The idea of using VBS enclaves is that we could run parts of the game in this isolated environment. That would mean that even if cheat developers somehow got kernel-mode code execution, they would still not be able to manipulate the code and data in this enclave.

A bit of skepticism

While it sounds like a good idea on paper, there are currently a few problems with the use of VBS enclaves:

- **Limited APIs** - You can't just take an existing program and tell Windows to run it in an enclave. It has to be a library (.DLL) specifically designed to run in such enclave which then needs a host process. There is a very limited set of APIs available (*libvcruntime*, *vertdll*, *UCRT*, *bcrypt*).
- **Performance** - Since virtualization is involved, there will always be some overhead, which in other use-cases is not that significant, but for a game trying to run logic or get data in/out of the enclave on each frame, this overhead is something that developers have to account for.
- **Security** - While in theory, the enclave should be absolutely impenetrable, in practice, cheat developers will always have the edge unless the system becomes completely locked down including the firmware. While it's true that the enclave is inaccessible to anything running *inside* the OS (since anything running inside the OS will run inside the virtualized environment), nothing is stopping cheat developers from writing a bootkit that will load before the OS even starts, therefore before any virtualization takes place. With a clever hook chain, they can gain complete control over the Hyper-V/VBS architecture itself (something I will be exploring in future blog posts).

Regardless of these issues, I decided to write a simple proof-of-concept project to test this idea out and to have something to experiment with in the future.

Getting started

Since VBS enclave functionality is quite new, the available sample projects and documentation are very sparse. In fact, I was not able to find any open-source project utilizing VBS enclaves, so the only starting point I could use was the official documentation and sample project.

Development tools required:

- Visual Studio 2022 version 17.9 or later
- Windows Software Development Kit (SDK) version 10.0.22621.3233 or later

Device/OS requirements:

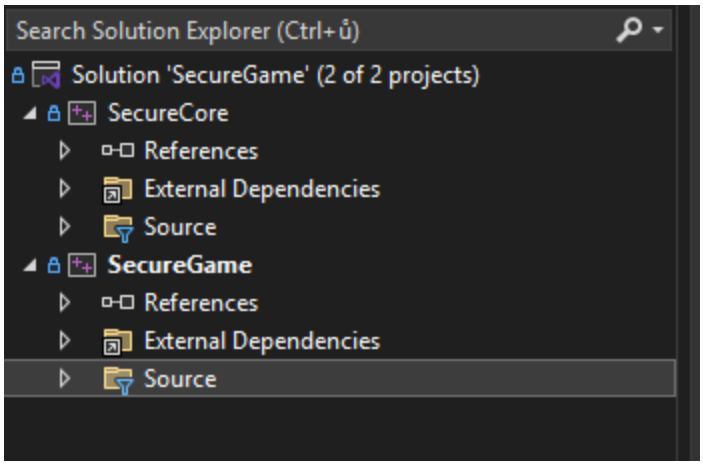
- Windows 11 or later or Windows Server 2019 or later
- VBS/HVCI must be enabled

- For debugging and running enclaves without production signing, test-signing must be enabled

I used VMware Workstation to run the latest version of Windows 11. Don't forget to enable nested virtualization in setting if you also do so (Virtualize Intel VT-x/EPT or AMD-V/RVI).

Since I wanted something extremely simple, I decided to write a game inspired by the classic Pong. The idea was to have a host process that would handle initialization, window creation, keyboard input and rendering, while the enclave would run the actual game logic.

I created a new solution with two projects: SecureGame, which would be the host process, and SecureCore, which would be the enclave itself.



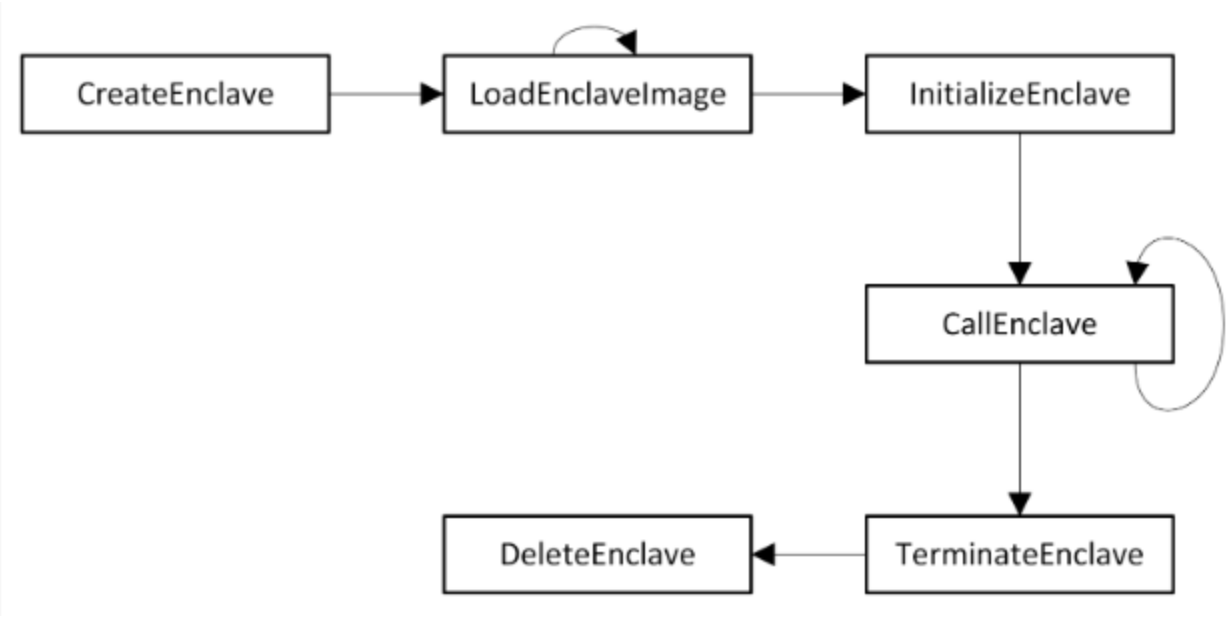
I used vcpkg to install SDL2 and then started implementing the game logic. There's nothing special about it; I ended up with a game loop like this:

```
void Game::Loop()
{
    constexpr int FPS = 240;
    constexpr int frameDelay = 1000 / FPS;

    bool running = true;
    while (running)
    {
        const Uint32 frameStart = SDL_GetTicks();

        SDL_Event event;
        while (SDL_PollEvent(&event))
        {
            if (event.type == SDL_QUIT)
                running = false;
        }

        SDL_SetRenderDrawColor(m_Renderer, 0, 0, 0, 255);
```

```
        SDL_RenderClear(m_Renderer);

        Tick();

        SDL_RenderPresent(m_Renderer);

        const int frameTime = SDL_GetTicks() - frameStart;
        if (frameDelay > frameTime)
            SDL_Delay(frameDelay - frameTime);
    }

    SDL_DestroyRenderer(m_Renderer);
    SDL_DestroyWindow(m_Window);
    SDL_Quit();
}
```

The interesting stuff

When the enclave is loaded, the host process can call any exported function from the enclave library using `CallEnclave()`.



Enclave lifecycle (source)

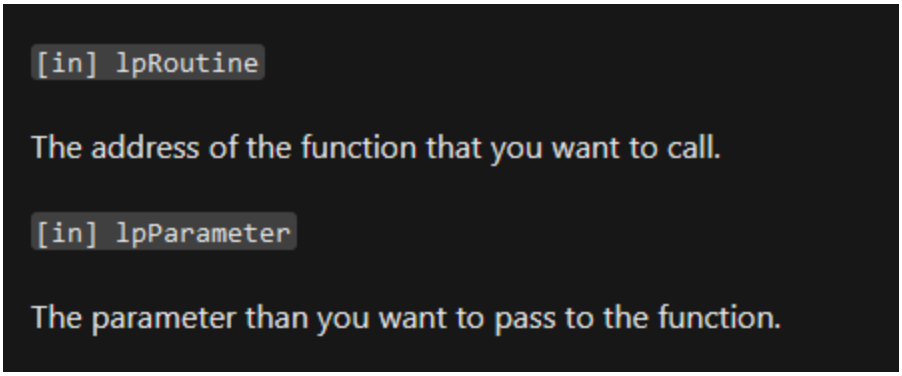
There are several ways we could approach using it:

1. Use it only for save/load - Store and load data from the enclave, keeping the loaded data in host process memory for the least amount of time possible and only for the time they are needed.
2. Supply all required input and run the entire game logic inside the enclave. The enclave then returns data needed for rendering, which is performed by the host process.

3. Put everything in the enclave - The `CallEnclave()` function can also perform a “reverse call” back to the host process [according to the documentation](). We could write a function that allows arbitrary function calls in the host process and then call this single function from within the enclave. This would allow us to put the entire game inside the enclave and work around its limitations.

While option 3 seems the most interesting, I went with option 2 to keep things simple.

Initially, I wondered whether you could pass a pointer to host process data into the enclave and access it directly. The documentation doesn’t mention whether the host process memory is accessible to the enclave at all, and to make matters worse, the sample code only passes data values, never pointers.



`CallEnclave()` (very helpful) documentation ([source]())

By trying it out, I discovered that the enclave **has access to the host process memory**, so you can pass pointers to data structures inside the host process. With this knowledge, I created a structure to be shared between the enclave and the host process.

```
typedef struct _TICK_DATA
{
    float DeltaTime;

    bool KeyW;
    bool KeyS;
    bool KeyUp;
    bool KeyDown;

    struct
    {
        float X;
        float Y;
        float Width;
        float Height;
    } LeftPaddle, RightPaddle, Ball;
```

```
    int LeftScore;
    int RightScore;
} TICK_DATA;
```

This structure would hold information about which keys are pressed on the current tick, the time elapsed since the last frame rendered, and the output from the enclave containing game object positions and score.

The enclave would then implement the game logic like this (peak coding performance, don't judge):

```
void Reset()
{
    Data::BallPositionX = static_cast<float>(WINDOW_WIDTH) / 2 - BALL_SIZE / 2;
    Data::BallPositionY = static_cast<float>(WINDOW_HEIGHT) / 2 - BALL_SIZE / 2;

    Data::BallVelocityX = (rand() % 2 == 0) ? BALL_SPEED : -BALL_SPEED;
    Data::BallVelocityY = (rand() % 2 == 0) ? BALL_SPEED : -BALL_SPEED;

    Data::State = Data::StateId::Running;
}

void Run(TICK_DATA* currentTick)
{
    const float deltaTime = currentTick->DeltaTime;

    if (currentTick->KeyW && Data::LeftPaddleY > 0)
        Data::LeftPaddleY -= PADDLE_SPEED * deltaTime;
    if (currentTick->KeyS && Data::LeftPaddleY < WINDOW_HEIGHT - PADDLE_HEIGHT)
        Data::LeftPaddleY += PADDLE_SPEED * deltaTime;
    if (currentTick->KeyUp && Data::RightPaddleY > 0)
        Data::RightPaddleY -= PADDLE_SPEED * deltaTime;
    if (currentTick->KeyDown && Data::RightPaddleY < WINDOW_HEIGHT - PADDLE_HEIGHT)
        Data::RightPaddleY += PADDLE_SPEED * deltaTime;

    Data::BallPositionX += Data::BallVelocityX * deltaTime;
    Data::BallPositionY += Data::BallVelocityY * deltaTime;

    if (Data::BallPositionY <= 0 || Data::BallPositionY + BALL_SIZE >= WINDOW_HEIGHT)
        Data::BallVelocityY = -Data::BallVelocityY;

    const bool ballInLeftPaddleYRange = Data::BallPositionY + BALL_SIZE >= Data::LeftPa
        Data::BallPositionY <= Data::LeftPaddleY + PADDLE_HEIGHT;
    const bool ballInRightPaddleYRange = Data::BallPositionY + BALL_SIZE >= Data::Right
        Data::BallPositionY <= Data::RightPaddleY + PADDLE_HEIGHT;

    if (Data::BallPositionX <= 20 + PADDLE_WIDTH &&
        Data::BallPositionX >= 20 &&
```

```
        ballInLeftPaddleYRange)
    {
        Data::BallPositionX = 20 + PADDLE_WIDTH;
        Data::BallVelocityX = -Data::BallVelocityX;
    }

    if (Data::BallPositionX + BALL_SIZE >= WINDOW_WIDTH - 20 - PADDLE_WIDTH &&
        Data::BallPositionX <= WINDOW_WIDTH - 20 &&
        ballInRightPaddleYRange)
    {
        Data::BallPositionX = WINDOW_WIDTH - 20 - PADDLE_WIDTH - BALL_SIZE;
        Data::BallVelocityX = -Data::BallVelocityX;
    }

    if (Data::BallPositionX <= 0)
    {
        Data::RightScore++;
        Data::State = Data::StateId::Reset;
    }
    if (Data::BallPositionX + BALL_SIZE >= WINDOW_WIDTH)
    {
        Data::LeftScore++;
        Data::State = Data::StateId::Reset;
    }
}

extern "C" __declspec(dllexport) void* CALLBACK GameTick(PVOID context)
{
    TICK_DATA* currentTick = static_cast<TICK_DATA*>(context);

    switch (Data::State)
    {
    case Data::StateId::Reset:
        Reset();
        break;
    case Data::StateId::Running:
        Run(currentTick);
        break;
    }

    currentTick->LeftPaddle.X = 20;
    currentTick->LeftPaddle.Y = Data::LeftPaddleY;
    currentTick->LeftPaddle.Width = PADDLE_WIDTH;
    currentTick->LeftPaddle.Height = PADDLE_HEIGHT;

    currentTick->RightPaddle.X = WINDOW_WIDTH - 20 - PADDLE_WIDTH;
    currentTick->RightPaddle.Y = Data::RightPaddleY;
    currentTick->RightPaddle.Width = PADDLE_WIDTH;
    currentTick->RightPaddle.Height = PADDLE_HEIGHT;
```

```
    currentTick->Ball.X = Data::BallPositionX;
    currentTick->Ball.Y = Data::BallPositionY;
    currentTick->Ball.Width = BALL_SIZE;
    currentTick->Ball.Height = BALL_SIZE;

    currentTick->LeftScore = Data::LeftScore;
    currentTick->RightScore = Data::RightScore;

    return nullptr;
}
```

And then the enclave function is called from within the game tick in the host process, and then the resulting game objects are rendered:

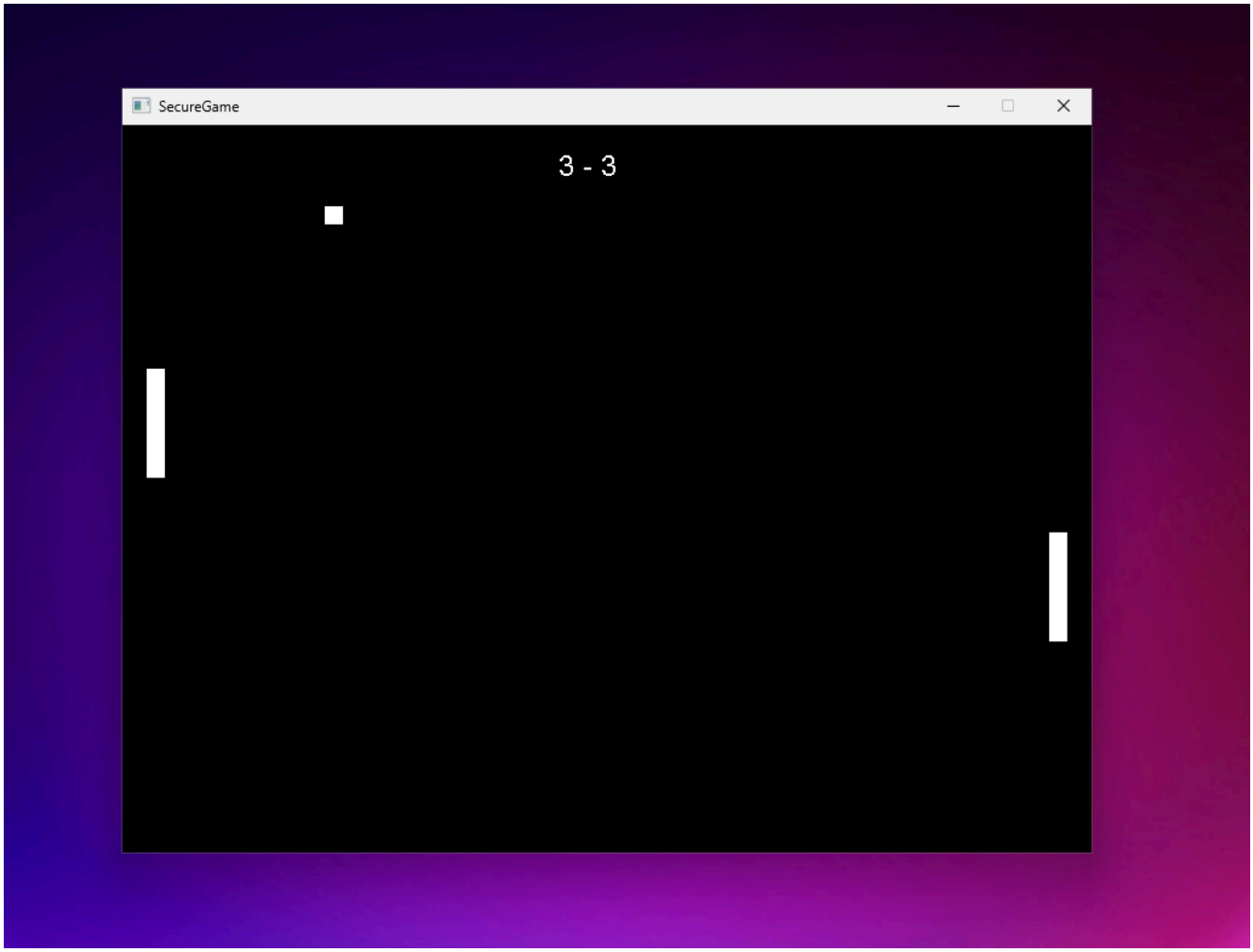
```
void Game::Tick()
{
    const Uint8* keystates = SDL_GetKeyboardState(nullptr);

    static Uint32 lastTime = SDL_GetTicks();
    const Uint32 currentTime = SDL_GetTicks();
    const float deltaTime = (currentTime - lastTime) / 1000.0f;
    lastTime = currentTime;

    TICK_DATA data;
    data.DeltaTime = deltaTime;
    data.KeyW = keystates[SDL_SCANCODE_W];
    data.KeyS = keystates[SDL_SCANCODE_S];
    data.KeyUp = keystates[SDL_SCANCODE_UP];
    data.KeyDown = keystates[SDL_SCANCODE_DOWN];

    PVOID returnValue = nullptr;
    if (!CallEnclave(Global::TickRoutine, &data, true, &returnValue))
    {
        char buffer[256];
        sprintf_s(buffer, "Failed to call enclave routine: %d", GetLastError());
        MessageBoxA(nullptr, buffer, "Error", MB_OK | MB_ICONERROR);
        return;
    }

    SDL_SetRenderDrawColor(m_Renderer, 255, 255, 255, 255);

    const SDL_Rect leftPaddle =
    {
        static_cast<int>(data.LeftPaddle.X),
        static_cast<int>(data.LeftPaddle.Y),
        static_cast<int>(data.LeftPaddle.Width),
        static_cast<int>(data.LeftPaddle.Height)
    };
```

```
    SDL_RenderFillRect(m_Renderer, &leftPaddle);

    const SDL_Rect rightPaddle =
    {
        static_cast<int>(data.RightPaddle.X),
        static_cast<int>(data.RightPaddle.Y),
        static_cast<int>(data.RightPaddle.Width),
        static_cast<int>(data.RightPaddle.Height)
    };
    SDL_RenderFillRect(m_Renderer, &rightPaddle);

    const SDL_Rect ball =
    {
        static_cast<int>(data.Ball.X),
        static_cast<int>(data.Ball.Y),
        static_cast<int>(data.Ball.Width),
        static_cast<int>(data.Ball.Height)
    };
    SDL_RenderFillRect(m_Renderer, &ball);

    char scoreText[32];
    sprintf_s(scoreText, "%d - %d", data.LeftScore, data.RightScore);
    RenderText(scoreText, WINDOW_WIDTH / 2 - 40, 20);
}
```

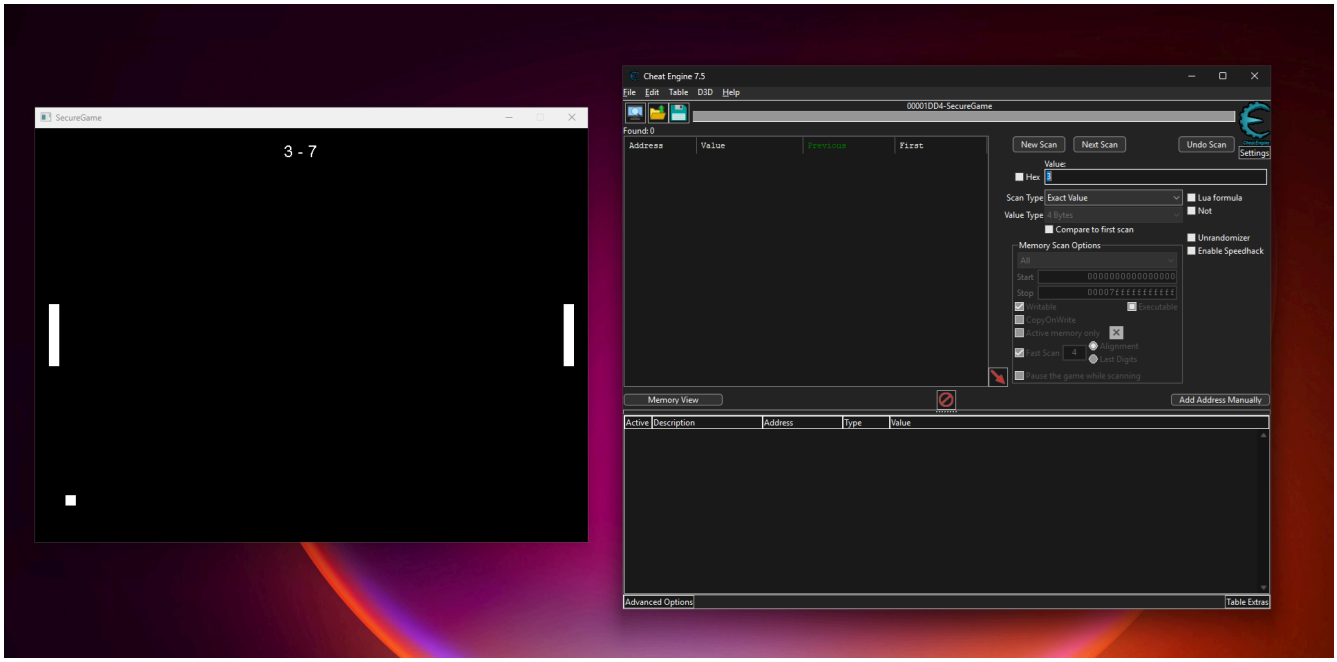
And that's it, now let's go test it out! Full source code available here.

Testing

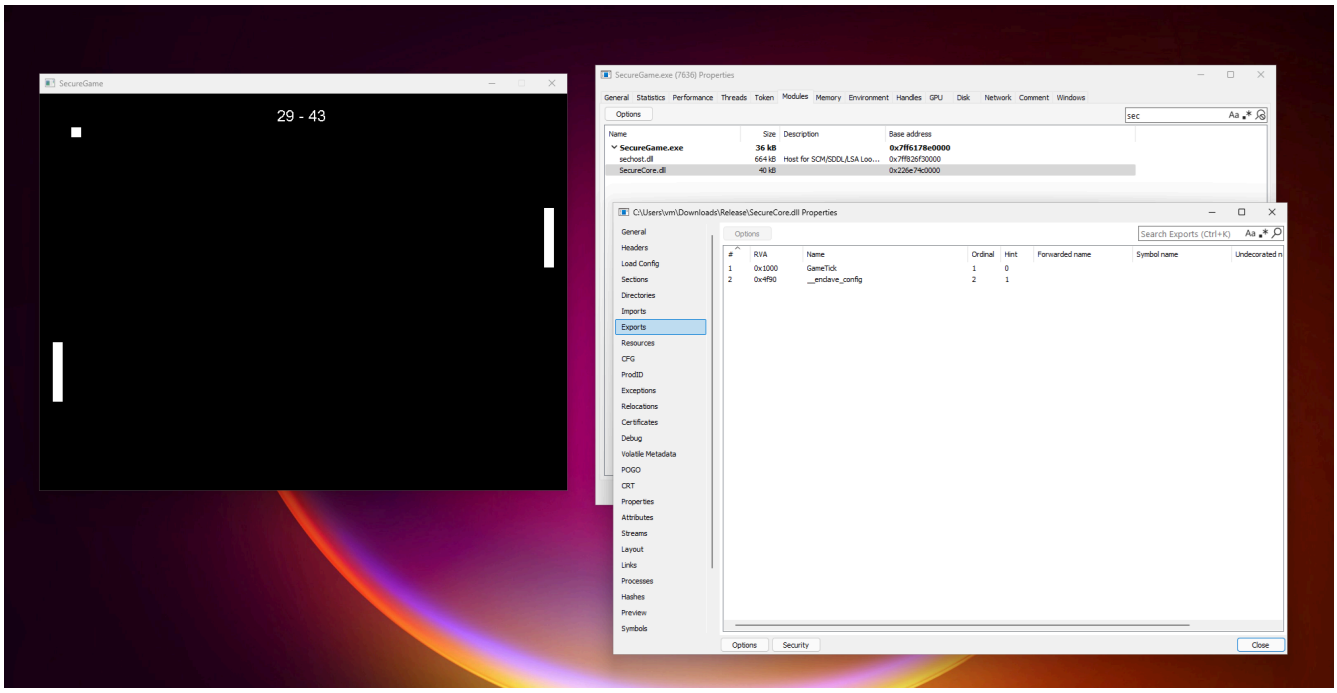
When you start the game, nothing will seem out of the ordinary (apart from the fact that it won't run without VBS enabled).



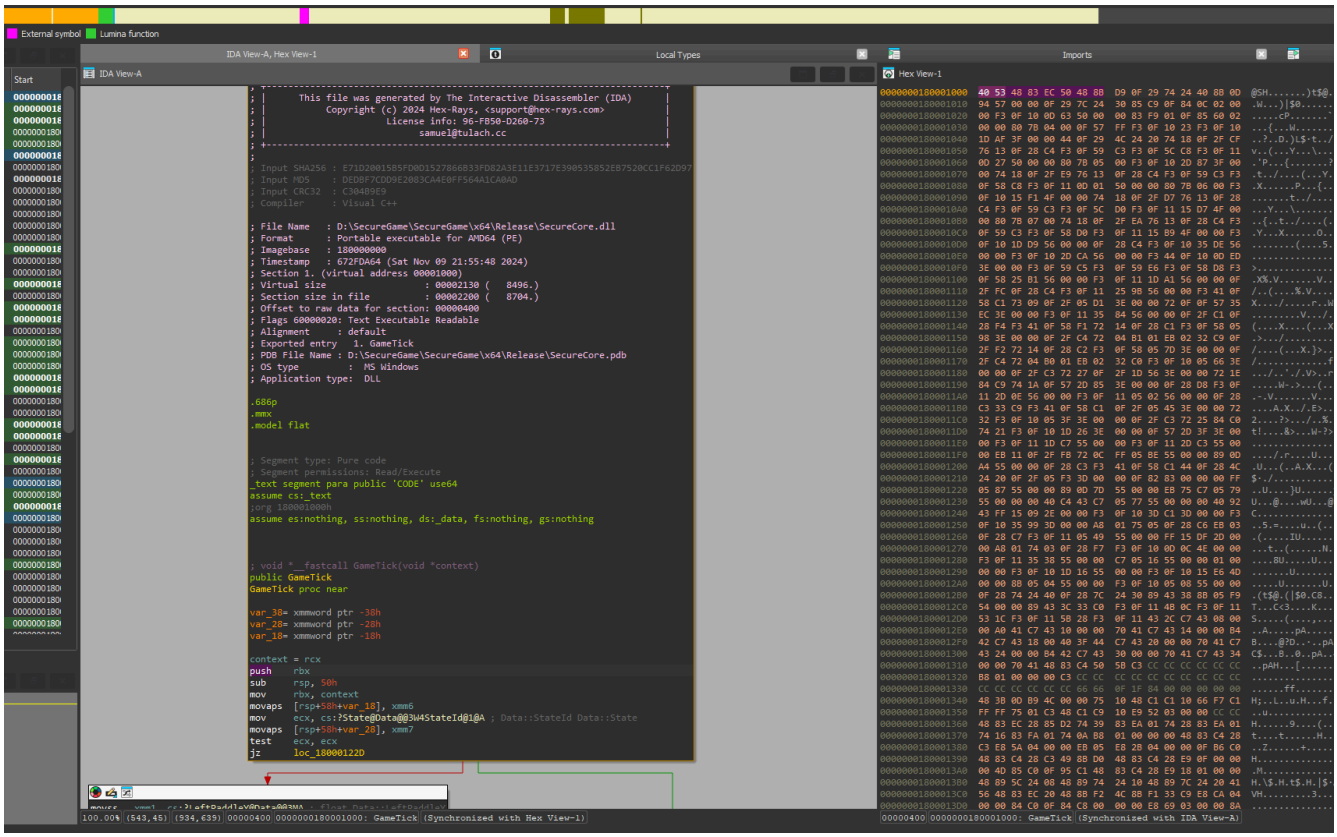
Let’s try to mess with it. Trying to change the score values using Cheat Engine won’t work. The score isn’t in the host process memory at all (or it’s there only for a very brief moment when it’s rendered on screen, but changing it won’t affect the actual score counter).



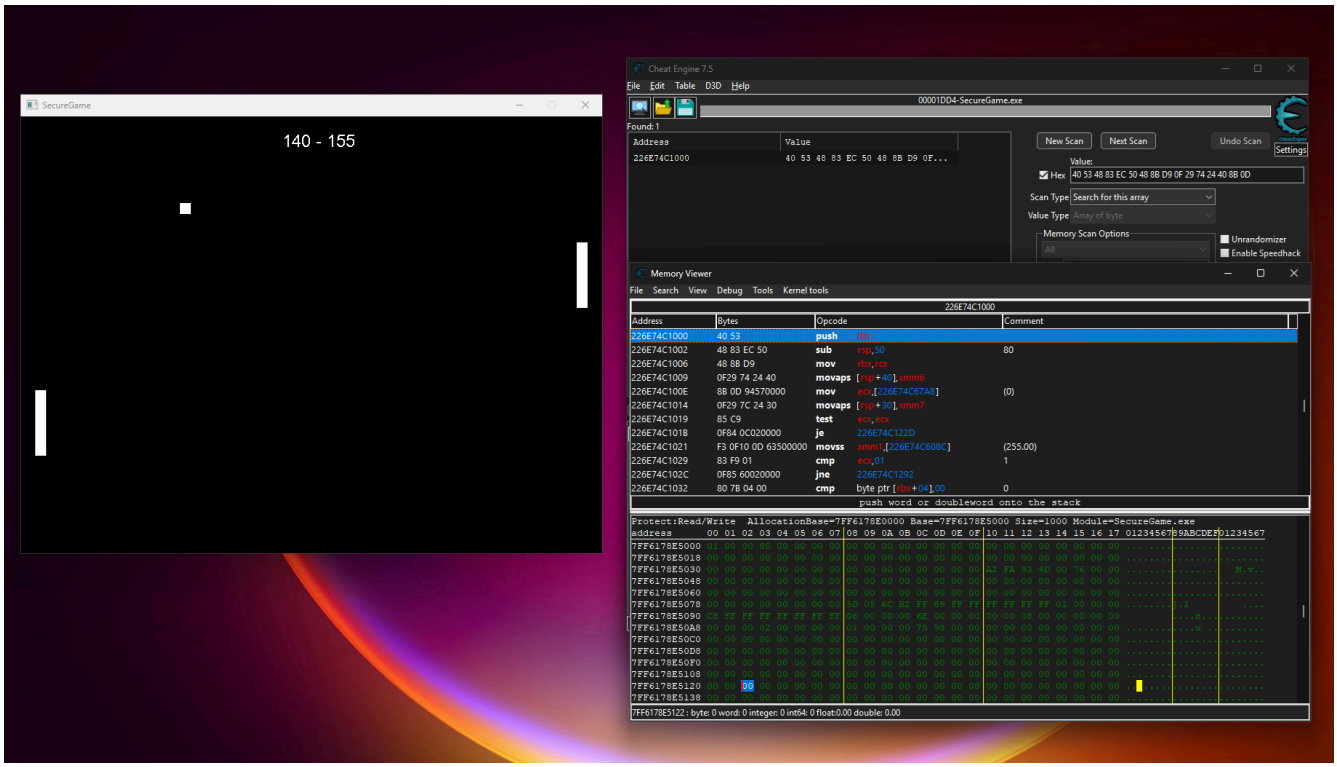
Ok, so what about trying to edit the code directly? When we look at the loaded modules of the process, we can actually see the loaded enclave DLL.



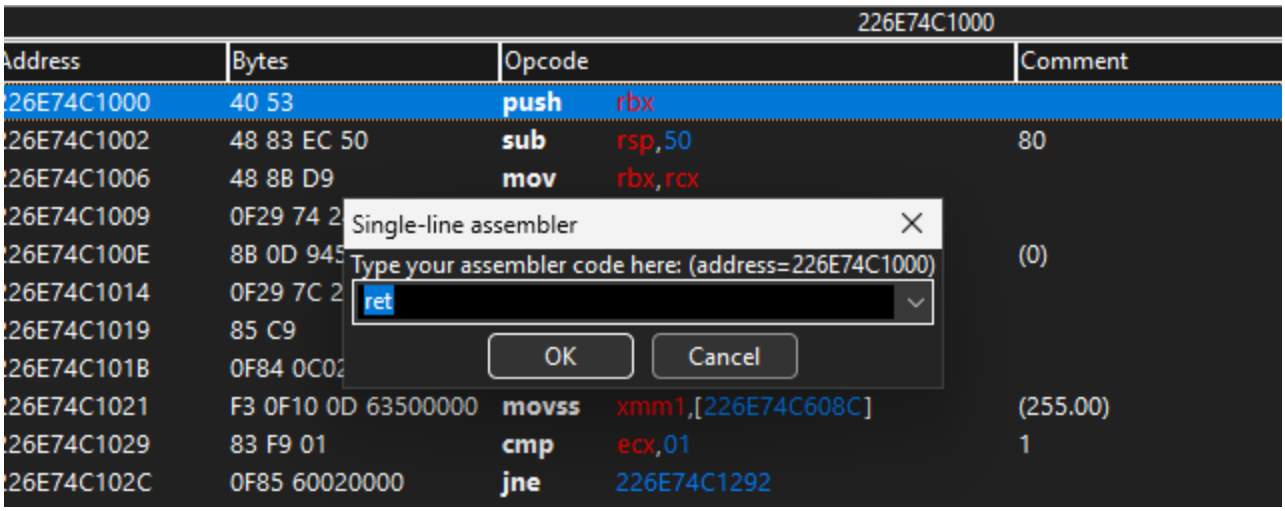
Does that mean we can just patch the module? Let's open `SecureCore.dll` in IDA and copy over the first bytes of the `GameTick()` function.



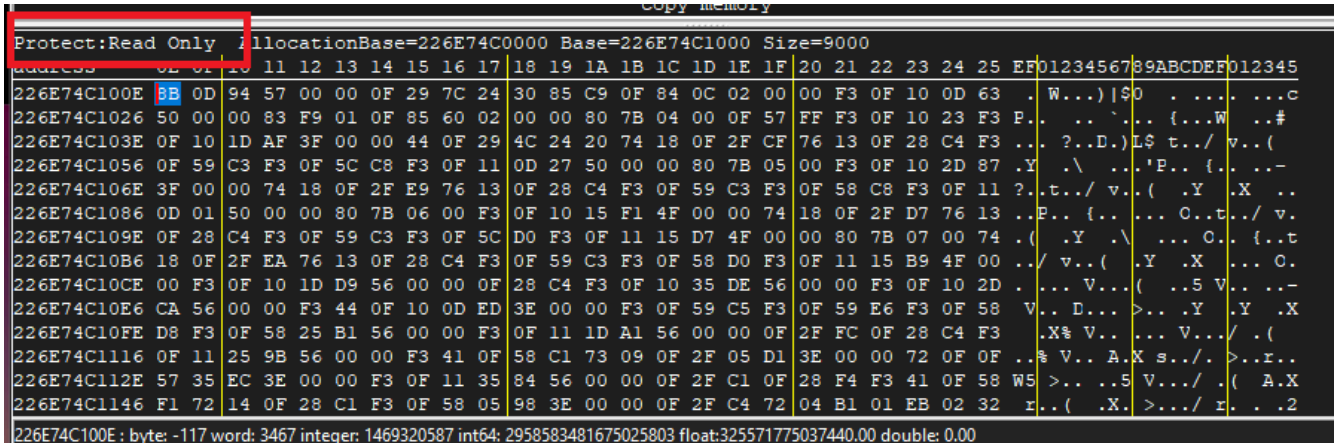
Then we can scan for the function using Cheat Engine.



Let's try to patch it by putting a return (0xC3) at the start of the function.



Aaaaand nothing. So what's going on? Well, first of all, let's check the memory protection.



It's actually just read-only. No execution permissions. That's because this is just a dummy image (most likely to prevent memory address conflicts). There's really no way we could mess around with the code or data in the enclave with anything we launch from within the OS.

Well, is there anything we **can** do then? Yes, obviously. Anything that's outside of the enclave is easily accessible. We can hook into the host process and edit the data returned by or sent to the enclave. We can also just inject Cheat Engine's speedhack DLL (which just hooks performance counters) and that will work too, since all the time calculations are done in the host process.

Conclusion

Due to the limitations mentioned above, it would take incredible effort to implement VBS enclaves into an actual game engine in a way that would be meaningful and not completely obliterate the game's performance. As such, I don't really see any game developers using them, not even accounting for the system requirements (Windows 11+, VBS enabled) since gamers might not be willing to reconfigure their systems just to play some game.

On top of that, while it would prevent any attempt to manipulate the game from programs or drivers loaded in the OS, experienced developers would have no issues getting around these restrictions by writing firmware apps that would manipulate the entire Windows bootchain.

There are two things that Microsoft could do that would instantly eliminate the vast majority of game cheating:

1. Work with OEMs to enforce strict boot environment code signing policies. Secure Boot is not sufficient (1, 2). The system would have to verify everything from the moment it's turned on until it's turned off. If it allows loading any user-created firmware code, it's over. However, being this strict would essentially mean locking the system down to Windows only, or at least making it very difficult to install alternative OSes (Linux), which no sane person can support.

2. While this would require enormous effort, they could introduce whole “process enclaves” where an entire process could run in a separate virtualized environment while still having access to standard NT APIs. I’ve read several security-related blog posts from them and I feel this sort of containerization is something they’re aiming for, so we’ll see.

Thanks for reading and have a nice day.