Analysis Report for: 5860E87D7E149FE6F87AEA83A35A7AAB-cleaned.cs

Overall Functionality

This C# code interacts heavily with unmanaged C++ code through P/Invoke calls. The primary visible functionality is a simple Windows Forms application ("PvZ Dex") with two buttons: "Infinity Sun" and "No Cooldown." The buttons appear to be designed to modify the memory of a target process (likely a game, judging by the names "Infinity Sun" and "No Cooldown," and references to "GameAssembly.dll"). The application finds the target process by name, opens it, locates a specific module ("GameAssembly.dll"), and then writes integer and boolean values to specific memory addresses within that module. This strongly suggests a memory-based trainer or cheat for a game.

Function Summaries

The code consists of numerous functions, many of which are P/Invoke declarations to Windows API functions and C++ Standard Library functions. Here's a summary of the more important functions:

* **`.new(ulong __unnamed000, void* _Where, __clr_placement_new_t __unnamed002)` and `.new(ulong _Size, void* _Where)`:** These overloaded functions appear to be custom memory allocation functions. They simply return the provided memory address `_Where`.

* **`.std.exception.{ctor}(...)`:** Constructor for the C++ `std::exception` class. Initializes the exception object.

* **`.std.exception.{dtor}(...)`:** Destructor for the C++ `std::exception` class. Cleans up the exception object.

* **`.std.exception.what(...)`:** Returns the error message from a `std::exception` object.

* **`.std.exception.__vecDelDtor(...)`:** A vector destructor for `std::exception`. Handles destruction of multiple exception objects.

* **`.std.bad_alloc.{ctor}(...)`:** Constructor for `std::bad_alloc` exception.

* **`.std.bad_alloc.__vecDelDtor(...)`:** Vector destructor for `std::bad_alloc`.

* **`.std.bad_alloc.{dtor}(...)`:** Destructor for `std::bad_alloc`.

* **`.std.bad_array_new_length.{ctor}(...)`:** Constructor for `std::bad_array_new_length` exception.

* **`.std.bad_array_new_length.__vecDelDtor(...)`:** Vector destructor for `std::bad_array_new_length`.

* **`.std.bad_array_new_length.{dtor}(...)`:** Destructor for `std::bad_array_new_length`.

* **`.std._Throw_bad_array_new_length()`:** Throws a `std::bad_array_new_length` exception.

* **`.GetProcessIdByName(char* processName)`:** Retrieves the process ID of a process given its name. Uses Toolhelp32Snapshot.

* **`.GetModuleHandleExW(uint processID, char* moduleName)`:** Gets the handle to a module within a specified process.

* **`.Is64BitProcess()`:** Checks if the current process is 64-bit.

* **`.PtrToStringChars(string s)`:** Gets a pointer to the characters of a managed string.

* **`.msclr.interop.details.GetAnsiStringSize(string _str)`:** Calculates the size needed for an ANSI representation of a managed string.

* **`.msclr.interop.details.WriteAnsiString(sbyte* _buf, ulong _size, string _str)`:** Writes a managed string to an ANSI buffer.

* **`.main()`:** The main entry point of the application.

* **`.msclr.interop.details.char_buffer.{ctor}, .release, .get, .{dtor}`:** Functions managing a character buffer.

* **`.smethod_1, smethod_2, smethod_3, ..., smethod_104, smethod_11, smethod_12 ...`:** A large number of internal utility functions that appear to perform tasks such as memory allocation, string manipulation and handling exceptions; mostly low-level operations and wrappers around unmanaged code.

* **`MainForm.MainForm()`:** Constructor for the main form.

* **`MainForm.method_0()`:** Disposes of resources held by the form.

* **`MainForm.patchButton_Click(...)`:** The event handler for the "Infinity Sun" button. This function is the core of the memory modification.

* **`MainForm.button1_Click(...)`:** The event handler for the "No Cooldown" button, similar in structure to `patchButton_Click`.

* **`MainForm.InitializeComponent()`:** Generated by the Forms Designer; sets up the UI controls.

* **`MainForm.Dispose(...)`:** Overridden Dispose method for proper resource cleanup.

Control Flow (Significant Functions)

* **`MainForm.patchButton_Click(...)` and `MainForm.button1_Click(...)`:** Both functions follow a similar pattern:
1. Get the process ID of the target process using `.GetProcessIdByName()`.
2. Open the target process using `.OpenProcess()`.
3. Get the module handle of "GameAssembly.dll" using `.GetModuleHandleExW()`.
4. Read memory locations using `.smethod_35()`. Note the hardcoded offsets (e.g., `+ 28885864L`, `+ 184UL`, `+ 164UL`, `+ 283UL`). These offsets indicate specific memory locations in the game where the values are written.
5. Write to the calculated memory address either using `.Write()` or `.Write()` for the respective button.

Data Structures

* Many structs are declared using `[NativeCppClass]` and `[StructLayout(LayoutKind.Sequential)]`, indicating that these are direct mappings to C++ structures. The sizes of these structures are specified, but the internal members are not shown (only private `int` or `long` members are visible).

* **`EHExceptionRecord`:** A structure related to exception handling in C++.

* **`gcroot`:** A structure that appears to manage a garbage-collected string within unmanaged code.

* **`tagPROCESSENTRY32W`:** Represents a process entry from the Toolhelp32Snapshot function.

* **`tagPROPVARIANT`:** A structure potentially used to represent variant data types.

* **`_s__ThrowInfo`:** likely contains information about the C++ exceptions being thrown

* Several other structures (`_Collvec`, `_Ctypevec`, `_Cvtvec`, `_GUID`, `tagRECT`, `_iobuf`, `_ldiv_t`, `_lldiv_t`, etc.) are used by the C++ interop functions. Their exact internal structure is hidden, but their sizes are defined.

Malware Family Suggestion

Based on the functionality, this code is highly suggestive of a **process memory modifying trainer/cheat**. It doesn't exhibit typical characteristics of other malware families like ransomware, spyware, or rootkits. The specific targeting of a game process ("GameAssembly.dll") and the modifications to memory addresses related to in-game resources ("Infinity Sun," "No Cooldown") point to its use as a cheat tool in a game environment. However, it's crucial to note that malicious actors could adapt similar techniques for other types of malicious activities, such as injecting code, manipulating system settings through memory, or covering tracks. Careful analysis of the obscured unmanaged C++ functions is needed for a definitive assessment of malicious intent. The presence of extensive exception handling could suggest an attempt to hide crashes or errors resulting from detection mechanisms.