Anti-Cheat Detections

I focused on detections that generate significant evidence of cheating. If you look at it yourself, you might encounter many additional detections much simpler in structure. Checking R5AC encoded strings might help you get an idea for some of them.

- Virtual Method Tables
- Control Flow Analysis
- Badware Forwarding

Virtual Method Tables

R5AC is ensuring that VMTP's are pointing within expected bounds, and that read-only VMT related data has not been tampered with.

Pointer Verification

Every time the script calls `sub_1DCDD1`, the virtual method table pointer (VMTP) for the filesystem interface is verified by `VTP_GetFilesystemInterface`. This function performs an integrity check to ensure the interface has not been tampered with or injected. There are numerous other variants of this VMTP related detection, all on different interfaces and virtual method tables, or object instances.

How the detection works

1. **VMTP location check**
   - The function calculates the address of `FilesystemInterfaceVMTP` and checks whether it lies within a valid PE section of the main game module.
   - If the pointer is **outside expected sections**, this indicates possible tampering or injection.
2. **Module and export verification**
   - It scans the loaded modules for `KERNEL32.DLL` and finds the export `K32GetMappedFileName`.
   - This function is used to retrieve the **file path of the module** where the VMTP resides.
3. **Violation reporting**
   - If the VMTP is outside a legitimate module section, `R5AC::PushViolation` is called.
   - This flags a potential anti-cheat violation.

> In more recent builds, R5AC has started verifying that the backing segment of the VMTP's linear address are having a plausible characteristics. it must contain IMAGE_SCN_CNT_INITIALIZED_DATA/IMAGE_SCN_MEM_READ

Conclusion:
`VTP_GetFilesystemInterface` is an anti-tamper routine that validates the location of the filesystem VMTP and ensures it points to legitimate code within the game’s module. Any deviation triggers a violation report, preventing injected or modified interfaces from being used.

Here is an example of how this detection might look like.

```
// Everytime sub_1DCDD1 is called, the check (VTP_GetFilesystemInterface) is executed.

char __fastcall sub_1DCDD1()
{
  FilesystemInterface = (__int64 *)R5::VTP_GetFilesystemInterface();
  *((_QWORD *)v2 + 16) = FilesystemInterface;
  gpFileSystemInterface[0] = FilesystemInterface;
  return 1;
}

void **R5::VTP_GetFilesystemInterface()
{
    _LIST_ENTRY* lpK32GetMappedFileName = nullptr;
    void* VmtTablePtr = R5::FilesystemInterfaceVMTP;
    void* VmtTablePtr2 = R5::FilesystemInterfaceVMTP;
    __int64 ImageBaseAddress = (__int64)NtCurrentTeb()->ProcessEnvironmentBlock->ImageBaseAddress;

    // Plaintext strings (replacing encrypted runtime decoding)
    const char ModuleNameStr[] = "KERNEL32.DLL";
    const char ExportFuncStr[] = "K32GetMappedFileName";
    const char InterfaceStr[] = "VTP_GetFilesystemInterface";

    if (*(WORD*)ImageBaseAddress == 0x5A4D) // Check for 'MZ' PE signature
    {
        _IMAGE_NT_HEADERS64* lpNTH = (_IMAGE_NT_HEADERS64*)(ImageBaseAddress + *(int*)(ImageBaseAddress + 0x3C));
        if (lpNTH->OptionalHeader.Magic == 0x20B) // PE32+
        {
            unsigned int NumberOfSections = lpNTH->FileHeader.NumberOfSections;
            char* lpFirstSectionContentStart = (char*)&lpNTH->OptionalHeader + lpNTH->FileHeader.SizeOfOptionalHeader;

            if (NumberOfSections && lpFirstSectionContentStart)
            {
```

```
            _IMAGE_SECTION_HEADER* dwTableRVA =
(_IMAGE_SECTION_HEADER*)((char*)R5::FilesystemInterfaceVMTP -
ImageBaseAddress);
            _DWORD* sectionHeaders = (_DWORD*)(lpFirstSectionContentStart +
8);
            unsigned int nSectionIdx = 0;

            while (true)
            {
               _IMAGE_SECTION_HEADER* dwSegmentRVA =
(_IMAGE_SECTION_HEADER*)(unsigned int)sectionHeaders[1];
               if (dwSegmentRVA <= dwTableRVA &&
                  (unsigned int)((_DWORD)dwSegmentRVA + *sectionHeaders) >
(unsigned __int64)dwTableRVA)
               {
                  break; // Found containing section
               }

               ++nSectionIdx;
               sectionHeaders += 10;

               if (nSectionIdx >= NumberOfSections)
               {
                  // Outside module section — use default
                  VmtTablePtr = VmtTablePtr2;
                  goto ON_DETECTED_OUTSIDE_INGAME_SECTION;
               }
            }
         }
      }
   }

ON_DETECTED_OUTSIDE_INGAME_SECTION:

   char lpMappedFileName[272] = {0};
   wchar_t NeedleForSomeModule[264] = {0};
   mbstowcs_s(nullptr, NeedleForSomeModule, ModuleNameStr,
sizeof(ModuleNameStr));

   // Search loaded modules
   _LIST_ENTRY* p_InMemoryOrderModuleList = &NtCurrentTeb()-
>ProcessEnvironmentBlock->Ldr->InMemoryOrderModuleList;
   _LIST_ENTRY* Flink = p_InMemoryOrderModuleList->Flink;
```

```
    if (Flink != p_InMemoryOrderModuleList)
    {
        while (_wcsicmp(NeedleForSomeModule, (_WCHAR*)Flink[5].Flink) != 0)
        {
            Flink = Flink->Flink;
            if (Flink == p_InMemoryOrderModuleList)
                goto ON_DETECTED;
        }

        _LIST_ENTRY* lpModuleBase = Flink[2].Flink;
        if (lpModuleBase)
        {
LABEL_22:
            if (LOWORD(lpModuleBase->Flink) == 0x5A4D) // 'MZ'
            {
                __int64 lpExportDirectory = (__int64)lpModuleBase +
SHIDWORD(lpModuleBase[3].Blink);
                if (*(WORD*)(lpExportDirectory + 24) == 0x20B) // PE32+
                {
                    unsigned int* v27 = (unsigned int*)(lpExportDirectory + 0x88);
                    if (!v27)
                        v27 = nullptr;

                    if (v27)
                    {
                        unsigned int nExportEnumIdx = 0;
                        _DWORD* v29 = (_DWORD*)((char*)lpModuleBase + *v27);
                        __int64 AddressOfFunctions = (__int64)lpModuleBase + (unsigned
int)v29[7];
                        __int64 AddressOfNames = (__int64)lpModuleBase + (unsigned
int)v29[8];
                        unsigned int NumberOfExports = v29[6];
                        __int64 AddressOfNameOrdinals = (__int64)lpModuleBase +
(unsigned int)v29[9];

                        while (nExportEnumIdx < NumberOfExports)
                        {
                            char* lpExportRecordMaybe = (char*)lpModuleBase +
*(unsigned int*)(AddressOfNames + 4i64 * nExportEnumIdx);
                            if (strcmp(lpExportRecordMaybe, ExportFuncStr) == 0)
                            {
                                unsigned int dwFunctionRVA =
*(_DWORD*)(AddressOfFunctions
                                    + 4i64 * *(unsigned __int16*)(AddressOfNameOrdinals +
2i64 * nExportEnumIdx));
```

```
                        lpK32GetMappedFileName =
(_LIST_ENTRY*)((char*)lpModuleBase + dwFunctionRVA);
                        break;
                    }
                    ++nExportEnumIdx;
                }
            }
        }
    }
  }

ON_DETECTED:

   // Get the module file name of where this VMTP is residing in.
   if (lpK32GetMappedFileName)
      ((void(__fastcall*)(__int64, void*, char*,
__int64))lpK32GetMappedFileName)(
         -1,
         VmtTablePtr2,
         lpMappedFileName,
         sizeof(lpMappedFileName)
      );

   // Report violation if needed
   R5AC::PushViolation(nullptr, 2, (__int64)VmtTablePtr2,
(__int64)lpMappedFileName);

   return &R5::FilesystemInterfaceVMTP;
}
```

Control Flow Analysis

R5AC is planting control flow enumeration helpers into important game mode.

Just because a module is signed, doesn't mean that R5AC will give it a pass.

It follows a strict whitelist which at the time of this writing, consisted of the following ranges.

r5apex_dx12.exe	(+1000 → +13d2000) ; '.text'
KERNEL32.DLL	(+1000 → +7f4bb) ; '.text'
ntdll.dll	(+1000 → +119f1e) ; '.text'
ntdll.dll	+11a000 → +11a592) ; 'PAGE'
ntdll.dll	(+11b000 → +11b1f9) ; 'RT'
steamnetworkingsockets.dll	(+1000 → +2f63a8) ; '.text
steam_api64.dll	(+1000 → +243de) ; '.text'
EOSSDK-Win64-Shipping.dll	(+1000 → +d8117c) ; '.text

Their whitelist is using a relatively primitive structure for describing what they deem a whitelisted memory range:

> struct WHITELISTED_REGION
>
> {
>
> u64 Begin;
>
> u64 End;
>
> };

> In more recent builds of**Apex Legends**, this list is now more complex to interpret ~~and seems to have switched to using the LDR directly, with some middle steps that i haven't figured out yet..~~ **EDIT**: actually, they still build a list of whitelisted regions. I was just too focused on other things and the hash calculation memes had initially confused me. Check S30 r5apex_dx12.exe+0x1D83E40 for the initializer, which is responsible for looking up the used API's and construct the initial list.

> In 2026, R5AC has switched to manual stack tracing via RtlCaptureContext + RtlLookupFunctionEntry + RtlVirtualUnwind.

> Additionally, respawn seems to have added some old code back, which was basically a broad check for return address spoofing. It first checks if the current retaddr entry that it previously captured using their manual stack unwind. It only proceeds with the actual heuristic if the memory interpreted as uint8 at retaddr-5 is not evaluated to 0xE8. In x64, this is a direct near CALL instruction using a signed, 32-bit displacement for affecting the program counter. So if it reads something that isn't 0xE8, it will proceed to run the following heuristic. It starts by doing a backwards scan, beginning at retaddr - 2. It then continuously keeps interating until a maximum of 6. It checks whether any code bytes during the backwards scan evaluate to 0xFF. If that is positive, it will try to filter out the REG field of the current instruction's MODRM byte, and if it doesn't find patterns resembling a FF/2 or FF/3 type of CALL instruction, it will fall through to the telemetry submission label,

> Additional heads up: If the currently scanned retaddr happens to be of value zero, it will jump directly to the telemetry submission label. I can't say if the telemetry is actually sent because in that label it does a couple further checks first before finally sending it.

An example of this instance of detection can be found in C_ViewRender::GetMostRecentClipTransform.

> ```
> __int64 C_ViewRender::GetMostRecentClipTransform(__int64 a1, char frameIndex)
> {
>   v152 = frameIndex;  // Prime RNG (seems unrelated to stack walker)
>   R5AC::Prime = 214013 * R5AC::Prime + 2531011;  // Early exit: check stack walker enablement
>   if (!R5AC::IsStackWalkerEnabled)
>     return *(_QWORD *)(a1 + 8i64 * v152 + 1155920);  __int64 stackFrames[2] = { 0 };
>   __int128 stackData[2] = { 0 };  unsigned short numFrames = R5AC::Imp::RtlCaptureSBT(1, 5, stackData);
> ```

```
    if (!numFrames)
        return *(_QWORD *)(a1 + 8i64 * v152 + 1155920);    // Compute simple
hash of the captured stack
    int hashA = 5381, hashB = 0;
    char* p = reinterpret_cast<char*>(stackData);
    for (size_t i = 0; i < 8 * numFrames; ++i)
    {
        unsigned char val = *p++;
        hashA = val + 33 * hashA;
        hashB = val + 65599 * hashB;
    }
    int stackHash = hashA ^ hashB;    // Lock to check hash
    MEMORY[0x7FFE8D571760](&R5AC::SoemLock, 0);
    char hasChanged =
R5AC::HashmapHasChangedData((__int64)&R5::HashedCodeExecutionFrames,
&stackHash);
    MEMORY[0x7FFE8D571920](&R5AC::SoemLock);    if (hasChanged != -1)
    {
        // If hash is known, return cached transform
        return *(_QWORD *)(a1 + 8i64 * v152 + 1155920);
    }    // Iterate captured frames
    for (unsigned int i = 0; i < numFrames; ++i)
    {
        __int64 frameAddr = *reinterpret_cast<__int64*>(stackData + i);
        if (!frameAddr)
            break;        // Check if frame is in whitelisted regions
        bool inWhitelist = false;
        for (int j = 0; j < R5AC::WhitelistedSectionCount; ++j)
        {
            _WHITELISTED_SEGMENT_* region = &R5AC::WhitelistedSections[j];
            if (frameAddr >= region->Begin && frameAddr <= region->End)
            {
                inWhitelist = true;
                break;
            }
        }        if (inWhitelist)
            continue;        // If not whitelisted, fetch mapped module and push
violation
        char moduleName[272] = { 0 };
        ((void(__fastcall*)(__int64, __int64*, char*, __int64))0)(-1, frameAddr,
moduleName, 260);        if (moduleName[0])
        {
            R5AC::PushViolation(nullptr, 1, frameAddr,
reinterpret_cast<__int64>(moduleName));
```

```
            return *(_QWORD *)(a1 + 8i64 * v152 + 1155920);
          }
      }    // If no violations, store new hash
      MEMORY[0x7FFE8D5790A0](&R5AC::SoemLock,
R5AC::WhitelistedSectionCount, numFrames, R5AC::WhitelistedSections);
      sub_2345E0(&R5::HashedCodeExecutionFrames, &stackHash);
      MEMORY[0x7FFE8D562C70](&R5AC::SoemLock);    return *(_QWORD *)(a1 +
8i64 * v152 + 1155920);
}
```

A look at the Callers of RtlCaptureSBT

Some cheat developers have actually been using the aforementioned functionality against apex. See, because the API respawn uses for stack walking is resolved into an unchecked pointer in .data, anyone can just swap it out and fake the back trace buffer, or even cheaper: pretend like nothing on the stack was unwindable. As if that wasn't enough, respawn also have decided to making calls by referencing it's value across the game's entire code base.



What makes this much worse is the fact that this vulnerable pointer is referenced by numerous critical game functions. It's literally a backdoor so handy that you don't even need to hook CreateMove anymore, for example. You can just swap this pointer, check if caller (return address) is **C_Input::CreateMove**, and and that's it! Well, you'd also have to reconstruct the arguments passed down to it if your hook cares about that. It definitely will if you are planning to use the sequence number argument to retrieve an usercmd pointer.

> Actually, you could detect this quite trivially by checking where these manually resolved API's point to. You might also be detected indirectly if they actually keep track of writable regions and diff them or something like that. I know that approach might be very effective for catching changes on read-only stuff. Maybe they're doing these or something else already.

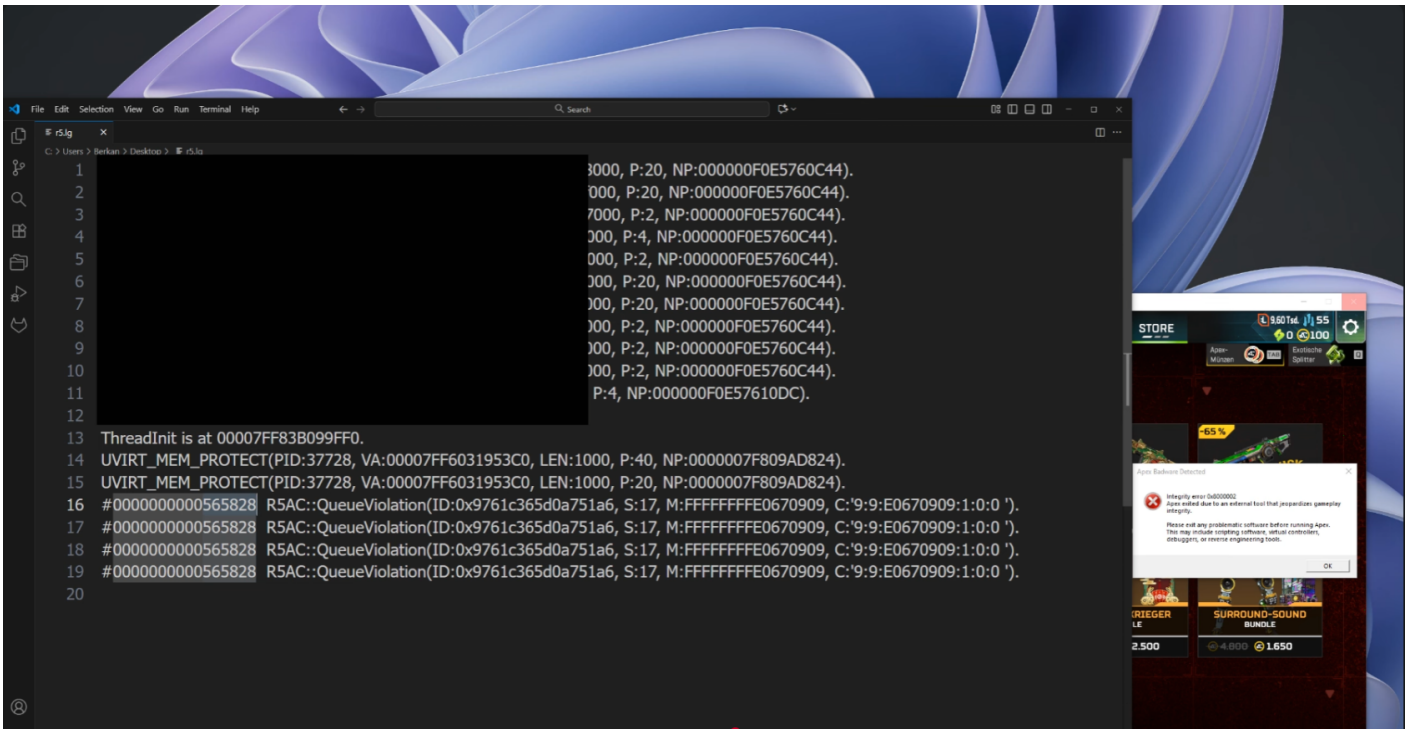
Above code was tested on retail apex legends, on both windows and linux. Since the flaw is in R5AC itself, platform mostly does not matter.

Badware Forwarding

I'm not entirely sure on who is doing the actual badware checking, i suspect it might be either Theia or R5AC.

While apex had still used EAC+Theia, there used to be a callback that inside the game that was called everytime a badware detection has fired. So the flow is you get the badware message, and in the background, Apex might send the violation possibly containing an identifier for the blacklisted tool that was detected.

I did a quick test by first neutralizing R5AC::PushViolation, and then monitoring it while simultaneously launching IDA Pro, which apparently was on their list of applications considered as badware. The result was that PushViolation was called multiple times nearly imemdiately after the blacklisted application was launched.



I should have tried launching cheat engine or some other blacklisted program and compare the c-string it passed to PushViolation. Sadly i have not done that so i can only speculate if they actually have a way of distinguishing the blacklisted application that was detected. It would certainly make sense though, purely from a detection perspective.

https://www.youtube.com/watch?v=Tcc6eExS7hk

It should be noted that this is one of the few functions that make use of the R5AC::PushViolation variant that, under the conditions where the game client currently has an active network channel to the game server, it will attempt to transmit the violation message directly via C_NetChan::SendNetMsg and C_NetChan::SendDatagram.