

# Attempting to Hack The FiveM Client

---

The final project specification asked students to pick one target for bug hunting. When we were asked this we thought of many things but the main thing that crossed our minds was [FiveM](#), a popular GTA V multiplayer modification client. One of the members in our group found bugs in several FiveM frameworks ranging from custom implementations to the second most popular framework, QBCore. But after all this, the target that fit best with the specification of this class was the FiveM client. Thus, we set out to see if we had what it takes to hack a open source project with 2900+ stars on github, hundreds of thousands of concurrent players, as well as hundreds of developers working to improve it. The repository containing all articles related to the client hacking can be found at our [Github repository](#).

## Introduction on Target

Before diving into the technical details of attacking the FiveM client, you are probably curious about what FiveM is. FiveM is a multiplayer game modification for the popular Rockstar Games product, Grand Theft Auto V (GTA 5). FiveM was developed by the team at CitizenFX who work to modify Rockstar Games' modern titles, all of which are open source and under the [same repository](#). Unlike most game modifications (known as mods in these communities), FiveM does not create one standalone experience. Instead, it gives developers an entire framework to interact with the GTA 5 process through the user client. The implementation uses the Citizen API that allows developers to use front-end runtimes/executables in Lua, Javascript, and even C# to make calls to the GTA 5 process. For example, a Javascript function `GetEntityHealth` will be passed in an ID of an in-game object and return its health, allowing developers to script out playing certain animations at certain health thresholds and more. Owners will host servers where they have these developers create scripts to interact with the GTA 5 process and FiveM client to provide users with a unique and seamless multiplayer experience. The above is a general overview of FiveM functionality, but there are a lot more moving parts to FiveM. The next section will cover what we focused on for the remainder of the project.

## The Attack Model

Since our codebase is massive, it makes the vulnerability analysis of the target very difficult. To minimize the scope of the analysis, we set up an attack model to restrict the context in which we attacked the application and treated exterior implementation details of the project as a black box. This was extremely helpful as it limited the scope in which we could research and attack the application to something more manageable for a team of beginner low-level bug hunters.

The attack model we established was a malicious server owner that sends users evil scripts that break out of the client runtime to run arbitrary code on the user system. This model relies on finding the C++ implementations of the Javascript V8 and Lua runtime and doing vulnerability analysis to discover bugs in them. During our assessment, we focused on the V8 Runtime because we had the most experience with Javascript, it is also the least researched of the runtimes, and it is relatively straightforward to set up.

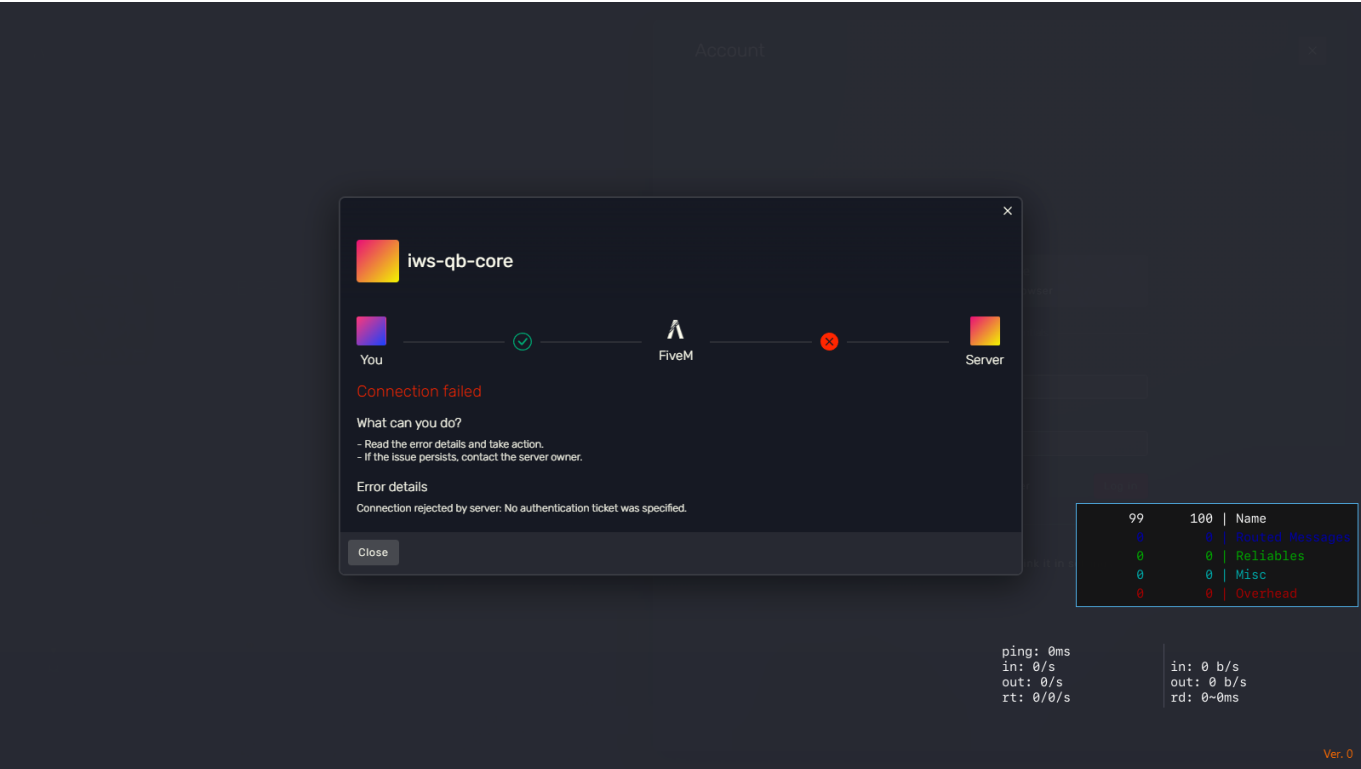
Before we could begin testing the Javascript runtime, we needed to first create our environment. For this we used IWS to create one Windows virtual machine to create a FiveM server running the default Cfx configuration and connected to it via any of the gaming capable desktops in the Cybersecurity room. With this setup, we created one manifest file to register commands in game and enabled them through our

server. When we ran the commands our code would run in the client side engine. This allowed us to make edits and deploy them to the client without restarting the server or client. The demo video for this can be found [here](#). With this set up, we can move into setting up a debug client.

## Debugging the client

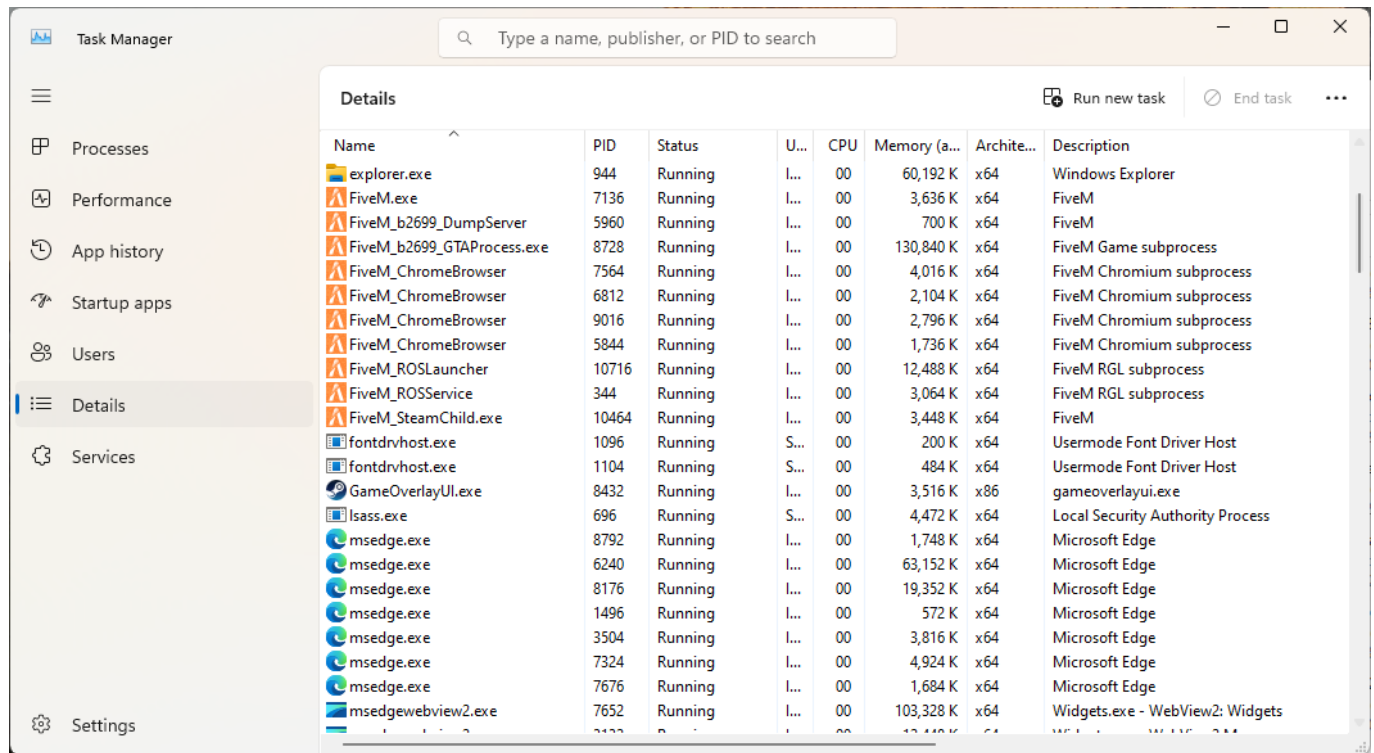
### Debug build

Initially, we expect this to be a simple task: starting or attaching x64dbg and we can start debugging. Since the source code is available [online](#), we successfully compiled a debug build from the repo. However, when running the debug build, we were not able to connect to the server. If the text box is true, the server we are hosting is not accepting and sending back authentication ticket to our debug client. We believe that the release build of FiveM has specific signature/key to link through Steam and FiveM signal server. Since we do not have them in our debug build, the connection process fails. After a bit of research, it seems that FiveM process is unable to communicate with Steam or the Source SDK process it uses for connecting.



### Official release build

So we changed to working with the official build version. Unfortunately, as soon as we use WinDbg or x64dbg, the process terminates shortly after, before the RIP could reach breakpoint. We suspect that the process can detect debugger on start, so we switch to attaching to the running (sub)processes.



At normal start, **FiveM.exe** produces the following processes (XXXX is 4-letter build number):

```
FiveM_bXXXX_GTAPProcess.exe
FiveM_bXXXX_DumpServer.exe
FiveM_ROSService
FiveM_ROSLauncher
FiveM_SteamChild.exe
FiveM_ChromeBrowser.exe
```

Given that our watch target is V8 engine and Lua runtime, we look for process that imports **citizen-scripting-v8client.dll**, **citizen-scripting-v8node.dll**, **citizen-scripting-lua54.dll**, **citizen-scripting-lua.dll**. Using Sysinternals Process Explorer, we found that only **FiveM\_bXXXX\_GTAPProcess.exe** matches.

Name	CPU	Private Bytes	Working Set	PID	Description	Company Name
steamwebhelper.exe	24.992 K	23,612 K	4216	4216	Steam Client WebHelper	Valve Corporation
FiveM_SteamChild.exe	< 0.01	52,420 K	25,532 K	10454	FiveM	Chx.re
GameOverlayUI.exe	0.17	40,636 K	18,748 K	8432	gameoverlayui.exe	Valve Corporation
msedge.exe	< 0.01	46,532 K	76,056 K	8176	Microsoft Edge	Microsoft Corporation
msedge.exe		2,100 K	8,176 K	1496	Microsoft Edge	Microsoft Corporation
msedge.exe		68,372 K	25,080 K	3504	Microsoft Edge	Microsoft Corporation
msedge.exe	< 0.01	13,016 K	35,480 K	7324	Microsoft Edge	Microsoft Corporation
msedge.exe		7,936 K	18,452 K	7676	Microsoft Edge	Microsoft Corporation
msedge.exe		13,500 K	15,220 K	8792	Microsoft Edge	Microsoft Corporation
msedge.exe	< 0.01	75,620 K	126,376 K	6240	Microsoft Edge	Microsoft Corporation
msedge.exe	< 0.01	98,364 K	26,760 K	7136	FiveM	Chx.re
FiveM.exe		2,732 K	11,436 K	5960	FiveM	Chx.re
FiveM_POSLauncher	< 0.01	167,656 K	35,380 K	10716	FiveM RGL subprocess	Chx.re
FiveM_POSService	< 0.01	72,044 K	22,500 K	344	FiveM RGL subprocess	Chx.re
FiveM_b2699_GTAProc	1.18	3,187,756 K	328,264 K	8728	FiveM Game subprocess	Chx.re
FiveM_ChromeBrowser	< 0.01	94,996 K	26,088 K	7564	FiveM Chromium subprocess	Chx.re
FiveM_ChromeBrowser		25,604 K	23,636 K	6812	FiveM Chromium subprocess	Chx.re
FiveM_ChromeBrowser		16,272 K	19,972 K	9016	FiveM Chromium subprocess	Chx.re
FiveM_ChromeBrowser		15,996 K	15,320 K	5844	FiveM Chromium subprocess	Chx.re
process.exe	2.02	25,516 K	49,476 K	16304	Sysinternals Process Explorer	Sysinternals - www.sysinter...
Taskmgr.exe	0.50	103,524 K	112,864 K	17696		

In few seconds after attaching to `FiveM_bXXXXX_GTAPProcess.exe`, exception such as `EXCEPTION_ACCESS_VIOLATION (0xc0000005)`, `EXCEPTION_ILLEGAL_INSTRUCTION (0xc000001d)` occurred and we could not continue. Looking at the disassembly, the address and instructions looks irregular, suggesting that certain kind of anti-debugger measures were implemented.

## Anti-anti-debug

Believing this is anti-debug issues, we found out ScyllaHide, an advanced user-mode anti-anti-debugger. Sadly, only the crash time is longer, which means there are multiple layers of anti-debugger. We also attempted enabling all measures provided by the library, but there are no significant changes.

C:\Users\IES\_Desert\_Eagle\AppData\Local\FiveM\FiveMAppData\cache\subprocess\FiveM\_b2699\_GTAProcess.exe - PID: 14640 - Thread: Main Thread 19624 - x64dbg

File View Debug Tracing Plugins Favourites Options Help Mar 4 2023 (TitanEngine)

CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols References Source Threads Handles Trace Snowman

```

00000000CD1F37AA70 add byte ptr ds:[rax],al
00000000CD1F37AA72 add byte ptr ds:[rax],al
00000000CD1F37AA74 add byte ptr ds:[rax],al
00000000CD1F37AA76 add byte ptr ds:[rax],al
00000000CD1F37AA78 00A8 371FC000 shr byte ptr ds:[rbx<CD1F37],1
00000000CD1F37AA7E add byte ptr ds:[rax],al
00000000CD1F37AA80 74 4D jnc CD1F37AA8E
00000000CD1F37AA82 1F
00000000CD1F37AA83 1F
00000000CD1F37AA84 CD 00
00000000CD1F37AA86 0000
00000000CD1F37AA88 93
00000000CD1F37AA89 BC 6087F9F7 mov esp,7FFB9870
00000000CD1F37AA8E add byte ptr ds:[rax],al
00000000CD1F37AA90 xor eax,cd0001
00000000CD1F37AA92 add byte ptr ds:[rax],al
00000000CD1F37AA94 add byte ptr ds:[rax],al
00000000CD1F37AA96 4E
00000000CD1F37AA98 37
00000000CD1F37AA9B 1F
00000000CD1F37AA9C AC 00
00000000CD1F37AA9E 0000
00000000CD1F37AA9F 0000
00000000CD1F37AAAB 80 6F mov al,6F
00000000CD1F37AAAD 0872 88 or byte ptr ds:[rdx<75],dh
00000000CD1F37AAAE D901 fld st(0),word ptr ds:[rcx]
00000000CD1F37AAAB 9A
00000000CD1F37AAAB 6C
00000000CD1F37AAAC F2
00000000CD1F37AAAB D5
00000000CD1F37AAAC 637A D9 movsd edi,dword ptr ds:[rdx<27]
00000000CD1F37AAAF 0105 000000CD add dword ptr ds:[cccc37AA85],eax
00000000CD1F37AA85 add byte ptr ds:[rax],al
00000000CD1F37AA87 0060 00
00000000CD1F37AA8A add byte ptr ds:[rax],al
00000000CD1F37AA8C add byte ptr ds:[rax],al
00000000CD1F37AA8E add byte ptr ds:[rax],al
00000000CD1F37AA90 add byte ptr ds:[rax],al
00000000CD1F37AA92 6F
00000000CD1F37AA93 jnn CD1F37AA9C

```

```

RAX 0000000000000005
RBX 0000000000000135
RCX 00000000CD1F37A878
RDX 0000000000000001
RBP 00000000CD1F37AA8F
RSP 00000000CD1F37AA83
RSI 00000000CD1F37A808
RDI 00000000CD1F37A878
R8 00000000CD1F37A808
R9 00000000CD1F37AA8F
R10 000002247E899B8A
R11 0000002243E8B004
R12 00000000CD1F37A800
R13 00000000CD1F37AF10
R14 00000000CD1F37AF00
R15 00000000CD1F37AD74
RIP 000002243A6B0686
RFLAGS 0000000000000102
ZF 0 OF 0 AE 0
CF 0 OF 0 OF 0
OF 0 TF 0 IF 1

```

LastError 00000000 (ERROR\_SUCCESS)  
 LastStatus 00000135 (STATUS\_DLL\_NOT\_FOUND)

Default (x64 fasmall) 5 | Unloaded

```

1: FCX 00000000CD1F37A878 00000000CD1F37A878
2: rdx 0000000000000001 0000000000000001
3: r8 00000000CD1F37A808 00000000CD1F37A808
4: r9 00000000CD1F37AA8F 00000000CD1F37AA8F
5: [rsp+28] 0000000000000005 0000000000000000

```

byte ptr ds:[rdi<CD1F37]=CD2004CAAF=???  
 CBAAL "A"  
 00000000CD1F37AA98

Dump 1 Dump 2 Dump 3 Dump 4 Dump 5 Watch 1 Locals Struct

```

Address Hex SA 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ.....Y-...
00007FF9B7590010 88 00 00 00 00 00 40 00 00 00 00 00 00 00 .....6.....
00007FF9B7590012 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00007FF9B7590014 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00007FF9B7590016 0E 1F BA 0E 00 84 09 CD 21 88 01 4C CD 21 54 68 .....1...11111111
00007FF9B7590018 68 73 20 70 72 6F 67 72 63 0E 20 61 6E 6E 6F .....is prog...canon
00007FF9B759001A 74 20 62 65 20 72 75 6E 20 69 6E 20 44 5F 20 32 ... be run in DOS
00007FF9B759001C 60 64 64 24 00 00 00 00 00 00 00 00 00 00 .....
00007FF9B759001E 1F BA 18 58 5B DB 75 EB 58 DB 75 EB 58 DB 75 EB ... (Due)Due (Due)Due
00007FF9B7590020 58 DB 75 EB 58 DB 75 EB 08 A4 75 EA 54 DB 75 EB ... (Due)Due (Due)Due
00007FF9B7590022 08 A4 75 EA 69 DA 75 EB 08 A4 71 EA DB DB 75 EB ... (Due)Due (Due)Due
00007FF9B7590024 08 A4 6A 58 5B 75 EB 08 A4 77 EA 54 DB 75 EB ... (Due)Due (Due)Due
00007FF9B7590026 52 69 63 68 5B DB 75 EB 00 00 00 00 00 00 00 ... (Due)Due (Due)Due
00007FF9B7590028 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....PE....
00007FF9B759002A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00007FF9B759002C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

```

00000000CD1F37A880 00000224003C003A
00000000CD1F37A882 000002247E899B80
00000000CD1F37A884 000000CD014001E2
00000000CD1F37A886 000002247E899B78
00000000CD1F37A888 000000CD01000042
00000000CD1F37A88A 000000CD01F37A8E00
00000000CD1F37A88C 005C003F00F005C0
00000000CD1F37A88E 005700C03000A043
00000000CD1F37A890 004F00440040E049
00000000CD1F37A892 005300C00530057
00000000CD1F37A894 0065007400730079
00000000CD1F37A896 005C003200300060
00000000CD1F37A898 006100170006400064
00000000CD1F37A89A 006C006400E002007F
00000000CD1F37A89C 000000CD0000009C
00000000CD1F37A89E 000000CD01F37AC0
00000000CD1F37A8A0 000000CD01F37AC7
00000000CD1F37A8A2 000000CD01F37AD0
00000000CD1F37A8A4 000000CD01F37AD7

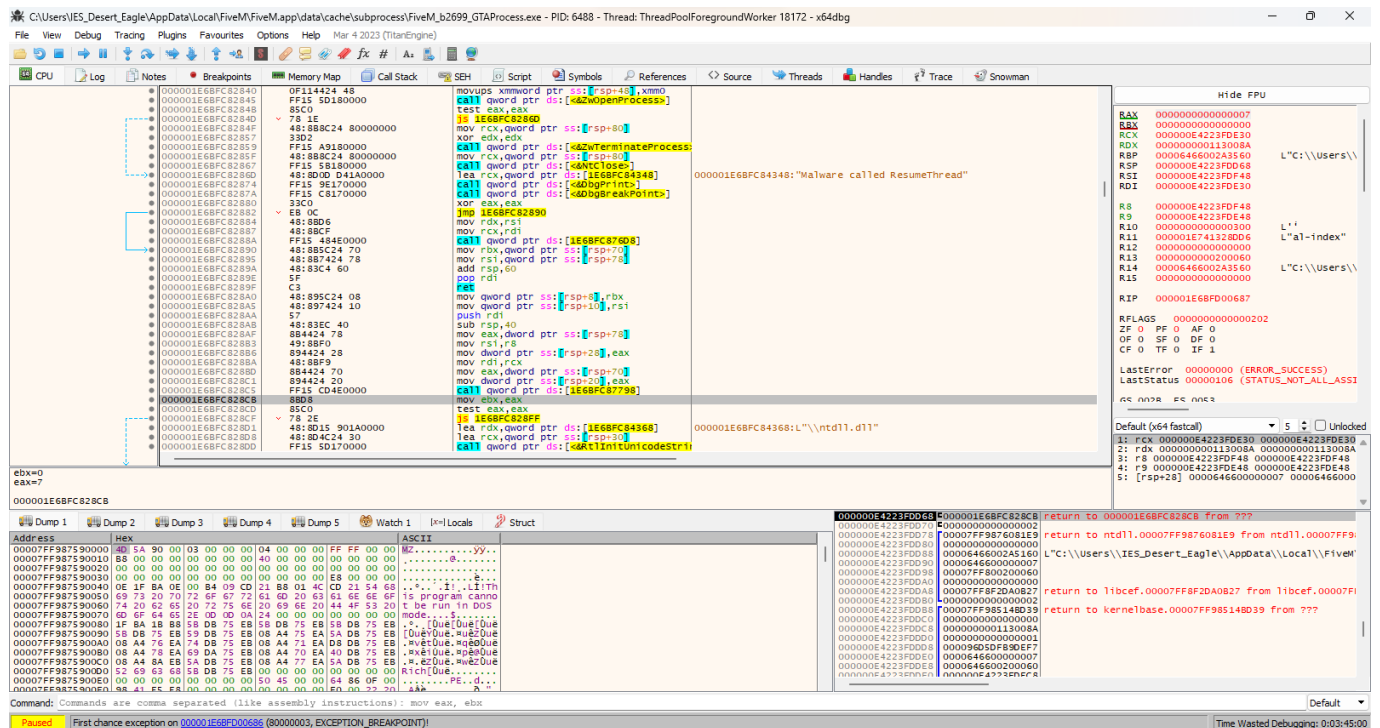
```

Command: Commands are comma separated (like assembly instructions): mov eax, ebx

First chance exception on 000002243A6B0686 (C0000005, EXCEPTION\_ACCESS\_VIOLATION)

Time Wasted Debugging: 0:03:42

We decided to look at the call stack during crashing. When clicking at several code addresses, they points to either middle of an instruction or invalid location. Even when it points to the start of an instruction, that instruction is not `jump`, `call`, and also does not affect RIP at all. Our initial thought is that anti-debugging is still working and has shifted the RIP.



At one thread, we found the existences of calls to `DbgPrint` and `DbgBreakPoint` inside `FiveM_bXXXX_GTAProcess.exe`, and we traced the import and found that they are from `ntdll.dll`. Technically, both of the calls are Windows APIs that execute `int 2D`, an interrupt to switch control to kernel debugger, similar to what the names suggest. But `int 2D` can also be used to anti-debug as it causes one byte shift after generating exception in several debuggers like OllyDbg. Suspecting this to be the issue, we set breakpoint at `ntdll.dll` and replaced `int 2D` with `nop` before `ntdll.dll` is loaded, but then the game windows no longer show up. With n successful results, we insisted on finding measures that is similar to `int 2D` in the source code instead.

## Anticlimactic

When we were searching anti-debug reference in the source code of FiveM, we came across this part at [code/client/citicore/SEHTableHandler.Win32.cpp, line 312-316](#):

```
static BOOLEAN WINAPI RtlDispatchExceptionStub(EXCEPTION_RECORD* record,
CONTEXT* context)
{
    // anti-anti-anti-anti-debug
    if (CoreIsDebuggerPresent() && (record->ExceptionCode == 0xc0000008 /*
|| record->ExceptionCode == 0x80000003*/))
    {
        return TRUE;
    }
}
```

`0xc0000008` exception code is `STATUS_INVALID_HANDLE`, meaning invalid handle. Based on the file name, an educated guess is that the program even has custom code to handle Win32 SEH (Structured Exception Handling), which is a mechanism that takes control and unwinds program in case an exception occurs in Windows. At this point, we realize the fact that GTA V itself is hard to debug, so the FiveM needs to implement several anti-debug in order to attach to FiveM.

Tracing `CoreIsDebuggerPresent` users, we found out that it is mostly used to workaround when debugging FiveM. For example, at [code/client/shared/Utils.cpp, line 186-197](#):

```
#if defined(_M_AMD64)
    if (CoreIsDebuggerPresent())
    {
        // thanks to anti-debug workarounds (IsBeingDebugged == FALSE),
        // we'll have to raise the exception to the debugger ourselves.
        // sadly, RaiseException (and, by extension, RtlRaiseException)
        // won't raise a first-chance exception, so we'll have to do such by hand...
        // again, it may appear things 'work' if using a non-first-chance
        // exception (i.e. the debugger will catch it), but VS doesn't like that and
        // lets some
        // cases of the exception fall through.

        RaiseDebugException(buffer.c_str(), buffer.length());
    }
    else
#endif
    {
        OutputDebugStringA(buffer.c_str());
    }
```

Here, they re-raise the exception just so that debugger would not steal the original exception. As it turns out, the error we received at the beginning might have been explicit exceptions raised by FiveM itself.

In conclusion, it is likely that our x64dbg and ScyllaHide have intervened the FiveM attaching process and caused issues. In fact, the `DbgBreakPoint` above is actually from ScyllaHide's hook library. Since we have been fighting *another debugger that has anti-anti-debug* and we did not have the same debugger as the developer, we eventually decided to rely on outside observation instead.

## NativeInvoke & InvokeNative

### NativeInvoke

Before we established an attack model, we read through the source code to get an understanding of the project. In our initial readthrough of the source code, we noticed one class `NativeInvoke` had some code that *looked* a lot like some bugs we've seen in class. But before that, what is `NativeInvoke`.

`NativeInvoke` is a class that includes a function `Invoke` that takes in an address to a GTA 5 method as well as the arguments to that method. Then the RAGE scripting engine will take these and execute the method in game. Afterwards it will return the response to the `NativeInvoke` which will in turn return it to the original caller. Let's see how it's implemented:



```

class NativeInvoke
{
public:
    template<uint32_t Hash, typename R, typename... Args>
    static inline R Invoke(Args... args)
    {
        NativeContext cxt;
        (cxt.Push(args), ...);

        static auto fn = rage::scrEngine::GetNativeHandler(Hash);
        if (fn != 0)
        {
            fn(&cxt);
        }

        if constexpr (!std::is_void_v<R>)
        {
            return cxt.GetResult<R>();
        }
    }
};

```

Okay, this is fine so far. How does the `NativeContext` look?

```

class NativeContext :
    public rage::scrNativeCallContext
{
private:
    // Configuration
    enum
    {
        MaxNativeParams = 16,
        ArgSize = 4,
    };

    // Anything temporary that we need
    uint8_t m_TempStack[MaxNativeParams * ArgSize];

public:
    inline NativeContext()
    {
        m_pArgs = &m_TempStack;
        m_pReturn = &m_TempStack; // It's okay to point both args and
return at                          // the same pointer. The game
should handle this.
        m_nArgCount = 0;
        m_nDataCount = 0;
    }
}

```

```

template <typename T>
inline void Push(T value)
{
    if (sizeof(T) > ArgSize)
    {
        // We only accept sized 4 or less arguments... that means no
        // double/f64 or large structs are allowed.
        throw "Argument has an invalid size";
    }
    else if (sizeof(T) < ArgSize)
    {
        // Ensure we don't have any stray data
        *reinterpret_cast<uintptr_t*>(m_TempStack + ArgSize *
m_nArgCount) = 0;
    }

    *reinterpret_cast<T*>(m_TempStack + ArgSize * m_nArgCount) = value;
    m_nArgCount++;
}
/* ... */
}

```

The `Push` method correctly checks argument sizes but our `TempStack` is continually pushed into without regards for its size. Meaning, if we are able to push enough arguments onto the stack such that it is greater than the size of the buffer then a buffer exploit is possible. Unfortunately, as much as this *looks* like a buffer overflow it is unfortunately not, well not really. In the `Invoke` function definition we see `static inline R Invoke(Args... args)`, especially the `Args... args` which is a function `parameter pack` that accepts zero or more function. In other words, this method will have a special definition for each number of possible arguments it can have *at compile time*. To exploit this, we must find any calls to `InvokeNative` with more than 16 arguments then we can actually cause an overflow. We'll go more into how we checked for this in the automated code analysis section.

## Invoke Native

So, since this `NativeInvoke` had a buffer overflow and was a static (compile time), we decided to focus on the frontend scripting API. In this, we found that they had a similar function `InvokeNative` that did the same thing as `NativeInvoke` but was dynamic and implemented differently. However, our hopes were quickly crushed when we found that this was patched and `previously exploited`. In this we decided to focus on V8 since Lua was mostly exploited by the attacker in the blog. Similarly, we decided to read through the source of both `InvokeNative` as well as all other `Citizen` functions.

How are these frontend functions implemented though? Essentially, there is a struct that holds keys that the names of these citizen functions and the values are the pointer to each respective function. Additionally, these functions are patched directly into the Runtime meaning they are callable via `Citizen.[FunctionName]`. With this, we looked through the code and scanned for memory bugs and our eyes found basically **nothing**. We also attempted to call these `Citizen` functions from the frontend with weird or edge case parameters but did not get any weird behavior or errors from the output logs. With these failures, we decided to move onto automated vulnerability analysis to push us in the right direction. At this point due to time and resource constraints we attempted to do automated static analysis.



# Automated Static Analysis

## Introduction

Since we determined dynamic analysis is not optimal given the circumstances, we can try our hand with some static analysis tools. We created two main strategies for bug hunting with our tool [Joern](#). Joern brings our code into nodes with additional associated information like callers, callees, args, etc. These nodes are interactable with the use of scala CLI/scripts to find bugs in the codebase. In our case, we had two strategies when looking for bugs in the program. The first involved scanning with Joern to find all places where [NativeInvoke](#) is used in the code and checking these locations to find where we can either find a place with 17+ arguments or dangerously passing user-controlled data. The second strategy was to create a *sink and source* model where our sources were all of the Citizen V8 functions while our sinks were any dangerous or potentially dangerous C++ functions that could introduce bugs.

## Finding NativeInvoke

### C++ Templates

At a very high level, C++ templates allow for generics for C++ classes and functions at compile time. So very unlucky for us, [NativeInvoke](#) uses a template so we cannot use [joern](#) easily. Normally in Joern, we can run the following command to get the callers of a function:

```
cpg.method.filter(_.name=="printf").caller.l
```

```

),
Method(
  id = 8529977L,
  astParentFullName = "code/components/tool-vehrec/src/VehicleRecordingTool.cpp:<global>",
  astParentType = "TYPE_DECL",
  code = ""static void DoFile(const boost::filesystem::path& path)

    static_assert(sizeof(VehicleRecordingEntry) == 32, "size");

    std::wstring fileNameStr = path.wstring();
    const wchar_t* fileName = fileNameStr.c_str();

    FILE* f = _wopen(fileName, L"rb");

    if (!f)
    {
        wprintf(L"couldn't open input file %s...\n", path.filename().c_str());
        return;
    }

    std::vector<VehicleRecordingEntry> entries;

    fseek(f, 0, SEEK_END);
    entries.resize(ftell(f) / sizeof(VehicleRecordingEntry));
    fseek(f, 0, SEEK_SET);

    fread(&entries[0], sizeof(VehicleRecordingEntry), entries.size(), f);

    fclose(f);

    auto bm = rage::five::pgStreamManager::BeginPacking();

    auto file = new(false) VehicleRecordingFile();
    file->SetBlockMap();
    file->SetEntries(entries.size(), &entries[0]);

    rage::five::pgStreamManager::EndPacking();

    std::wstring outFileName(fileName);
    outFileName = outFileName.substr(0, outFileName.length() - 3) + L"yvr";

    f = _wopen(outFileName.c_str(), L"wb");

    if (!f)
    {
        printf("... couldn't...",
columnNumber = Some(value = 1),
columnNumberEnd = Some(value = 1),
filename = "code/components/tool-vehrec/src/VehicleRecordingTool.cpp",
fullName = "DoFile",
hash = None,
isExternal = false,
lineNumber = Some(value = 46),
lineNumberEnd = Some(value = 109),
name = "DoFile",
order = 3,
signature = "void DoFile (ANY)"
    },
Method(
  id = 8530212L,
  astParentFullName = "code/components/tool-vehrec/src/VehicleRecordingTool.cpp:<global>",
  astParentType = "TYPE_DECL",
  code = ""static void Run(const boost::program_options::variables_map& map)

    if (map.count("filename") == 0)
    {
        printf("Usage:\n\n  fivem tool:vehrec *.rrr...\n");
        return;
    }

    auto& entries = map["filename"].as<std::vector<boost::filesystem::path>>();

    for (auto& filePath : entries)
    {
        DoFile(filePath);
    }

""
),

```

Since Joern doesn't know about these generics and is matching with strings of functions and signatures to each other this means we actually cannot get the callers of our function.

```

joern> cpq.method.filter(_.fullName=="NativeInvoke.Invoke").caller.l
res14: List[Method] = List()
joern> []

```

A workaround for this was to write a custom scala script to go iterate through all the base methods in the program and recursively iterate through the methods of their callers and return any point where we find a `NativeInvoke` call in the code. The code looked like the following:

```
def getInvokeCalls(max: Int=5): Unit = {
  def addRecurse(cur: Traversal[Method], count: Int){
    cur.foreach({res=>
      if(res.size > 0 && count<max){
        if(res.code.contains("NativeInvoke")){
          print("NativeInvoke @ ")
          print(res.filename)
          print(":")
          print(res.lineNumber)
          println()
        }
        addRecurse(res.callee, count+1)
      }
    })
  }
  addRecurse(cpg.method, 0)
}
```

And running it in the code we get the following output:

```
joern> getInvokeCalls()
NativeInvoke @ code/components/extra-natives-five/src/NuiAudioSink.cpp:Some(1940)
NativeInvoke @ code/components/scripthookv/src/VishCompat.cpp:Some(121)
NativeInvoke @ code/components/loading-screens-five/src/LoadingScreens.cpp:Some(71)
NativeInvoke @ code/components/loading-screens-five/src/LoadingScreens.cpp:Some(71)
NativeInvoke @ code/components/loading-screens-five/src/LoadingScreens.cpp:Some(354)
NativeInvoke @ code/components/scrbind-formats/src/Drawable.cpp:Some(269)
NativeInvoke @ code/components/lovely-script/src/Lovely.cpp:Some(67)
NativeInvoke @ code/components/lovely-script/src/Lovely.cpp:Some(139)
NativeInvoke @ code/components/lovely-script/src/Lovely.cpp:Some(67)
NativeInvoke @ code/components/gta-net-ny/src/HostSystemNY.cpp:Some(24)
NativeInvoke @ code/components/gta-net-ny/src/HostSystemNY.cpp:Some(71)
NativeInvoke @ code/components/extra-natives-rdr3/src/ShapeTestNatives.cpp:Some(62)
NativeInvoke @ code/components/citizen-level-loader-rdr3/src/LevelLoader.cpp:Some(60)
NativeInvoke @ code/components/gta-net-five/src/MumbleVoice.cpp:Some(605)
NativeInvoke @ code/components/gta-net-five/src/MumbleVoice.cpp:Some(790)
NativeInvoke @ code/components/gta-net-five/src/MumbleVoice.cpp:Some(814)
NativeInvoke @ code/components/extra-natives-rdr3/src/DoorExtraNatives.cpp:Some(58)
NativeInvoke @ code/components/citizen-level-loader-five/src/LevelLoader.cpp:Some(68)
NativeInvoke @ code/components/citizen-devtools/src/ResourceTimeWarnings.cpp:Some(30)
NativeInvoke @ code/components/scripting-gta/src/ScriptEngine.cpp:Some(96)
```

Now when we individually look at all of these calls we do not find anything out of the ordinary. Therefore, we moved to our second strategy.

## Mapping Sinks to Sources

The next strategy involves us declaring two lists, one for our sources (Citizen functions) and the other for our sinks (dangerous C++) functions. This part is simple enough, we write a script to iterate through all our

source functions and recursively search through their callees for any sinks. If we find one we print the call stack to get to it as well as where we can find it. The script is below:

```
def sink2src(max: Int=5) {
    val sinks = List("printf", "memcpy",
"scanf", "free", "Delete", "strcpy", "malloc", "calloc", "execve", "sprintf", "vspr
intf", "gets", "fgets", "read")
    val sources =
List("V8_Trace", "V8_SetTickFunction", "V8_SetEventFunction", "V8_SetCallRefFu
nction", "V8_SetDeleteRefFunction", "V8_SetDuplicateRefFunction", "V8_Canonica
lizeRef", "V8_MakeFunctionReference", "V8_GetTickCount", "V8_InvokeNative<Stri
ngHashGetter>", "V8_InvokeNative<IntHashGetter>", "V8_InvokeNativeRaw", "V8_Sn
ap", "V8_StartProfiling", "V8_StopProfiling", "V8_SetUnhandledPromiseRejection
Routine", "V8_SubmitBoundaryStart", "V8_SubmitBoundaryEnd", "V8_SetStackTraceR
outine", "V8_GetPointerField<V8MetaFields::PointerValueInt>", "V8_GetPointerF
ield<V8MetaFields::PointerValueFloat>", "V8_GetMetaField<V8MetaFields::Point
erValueInt>", "V8_GetMetaField<V8MetaFields::PointerValueFloat>", "V8_GetMeta
Field<V8MetaFields::PointerValueVector>", "V8_GetMetaField<V8MetaFields::Ret
urnResultAnyway>", "V8_GetMetaField<V8MetaFields::ResultAsInteger>", "V8_GetM
etaField<V8MetaFields::ResultAsLong>", "V8_GetMetaField<V8MetaFields::Result
AsFloat>", "V8_GetMetaField<V8MetaFields::ResultAsString>", "V8_GetMetaField<
V8MetaFields::ResultAsVector>", "V8_GetMetaField<V8MetaFields::ResultAsObjec
t>", "V8_GetResourcePath")

    def recurseSearch(cur: Traversal[Method], count: Int): Boolean = {
        cur.filter(!_.name.contains("<operator>")).foreach { res =>
            if (res.size > 0 && count < max) {
                if (sinks.contains(res.name)) {
                    println(s"${res.method.name} @
${res.filename}:${res.lineNumber}")
                    println("Call Stack:")
                    println(s"  -> ${res.name}")
                    return true
                } else {
                    val found = recurseSearch(res.callee, count + 1)
                    if (found) {
                        println(s"  -> ${res.name}")
                        return true
                    }
                }
            }
        }
    }
    false
}

cpg.method.filter{method=>sources.contains(method.name)}
    .foreach{res=>
        var found = recurseSearch(res, 0)
        if(found){
            println(s"  -> ${res.name}")
        }
    }
```

```
}
}
```

The output of this script finds a couple **free** and many **Delete** but again, we did not find anything that stood out:

```
free @ code/components/citizen-scripting-mono/deps/server/include/glib.h:Some(148)
Call Stack:
  -> free
  -> fwFree
  -> V8_CanonicalizeRef
  -> V8_CanonicalizeRef
Delete @ <empty>:None
Call Stack:
  -> Delete
  -> V8_StopProfiling
  -> V8_StopProfiling
Delete @ <empty>:None
Call Stack:
  -> Delete
  -> V8_Snap
  -> V8_Snap
free @ code/components/citizen-scripting-mono/deps/server/include/glib.h:Some(148)
Call Stack:
  -> free
  -> fwFree
  -> V8_CanonicalizeRef
  -> V8_CanonicalizeRef
Delete @ <empty>:None
Call Stack:
  -> Delete
  -> V8_StopProfiling
  -> V8_StopProfiling
Delete @ <empty>:None
Call Stack:
  -> Delete
  -> V8_Snap
  -> V8_Snap
free @ code/components/citizen-scripting-mono/deps/server/include/glib.h:Some(148)
Call Stack:
  -> free
  -> fwFree
  -> V8_CanonicalizeRef
  -> V8_CanonicalizeRef
Delete @ <empty>:None
Call Stack:
  -> Delete
  -> V8_StopProfiling
  -> V8_StopProfiling
```

With that, we exhausted the time and ideas we had for static analysis and presented our findings to the class. However, there are still ideas we didn't try that will definitely be interesting if we attempt to attack this target in the future.

## Future Plans

For the future, we hope to employ more dynamic techniques like taint analysis and fuzzing. Specially, we believe fuzzing the target would bring the most success. To fuzz this target we would have two options.

The first is to fuzz the V8Runtime directly by writing a harness to interact with the DLL and execute our Javascript. The fuzzer would be grammar based to generate valid Javascript and call the Citizen functions with the correct arguments. In this example, we would take our crashes and use the same input in our previous manual set up to debug what caused the crashes and if it led to anything exploitable. Alternatively we could do the same thing but create a custom snapshot based fuzzer that uses the FiveM server API to update the Javascript for each mutation and load the client at the point it finished loading into the game and executes the script. We would verify crashes the same way as our other fuzzer.

## Final Remarks

With this have finished cataloging our assessment of the target and concluded a few things. *Attacking large targets is hard.* If we were to redo this whole project we would pick a smaller target we could really dive into and get to understand entirely. *Time is key.* Good vulnerability research takes time and if we had more time and experience we could have evaluated this target better. *Pick something easy to debug.* Debugging is very important, and when you can't properly debug it sucks. Picking a target that had less debug protections would have made it a lot easier. All in all, this was a fun project to get to know how real world vulnerability research is done and what we can expect from real life targets.