390R Final Presentation - Checkpoint 1

Target Overview

FiveM is a popular modification (mod) for the game Grand Theft Auto V. Essentially it allows players to host custom servers with modded assets, scripted behavior, as well as other custom features. FiveM Client gives players a GUI client to browse, connect to, and play servers. The FiveM server allows developers to create scripts that interact with the FiveM framework to set up custom behavior to change the gameplay of the Grand Theft Auto V experience. When a player loads into a server the server will send scripts for the client to download and execute the Lua scripts on the client. There are several other features like an HTTP client, client side browsers (sadly up-to-date), as well as a console (which might be helpful in dynamic testing).

The feature of most interest is the script execution from server to client which would allow for the possibility of remote code execution on the client from joining a server. There have already been vulnerabilities in the past from this feature and we can use other features like the FiveM console for dynamic testing before hand. The target is very complicated and has a very large attack surface but we will focus our project on what is most easily testable and documented to do some exploit hunting.

Debug Environment

FiveM source code is publicly available on GitHub. Thanks to that, a debug build can be easily acquired by installing dependencies and compiling the project. The instructions followed to build FiveM and install the dependencies are found in the repository here. The build folder contains both object (`exe` and `dll`) and symbol (`pdb`/Program Database) files.

Two Windows debuggers are setup for this project: x64dbg and WinDbg Preview. x64dbg is selected due to the capabilities of source code debugging and PDB parsing. Meanwhile, WinDbg Preview is chosen as a backup in case we could not attach using x64dbg.

Map Out the Codebase

As stated earlier, the codebase for FiveM is public, meaning there is a lot of source code that may or may not be meaningful for an attacker reading the code. To start the process of mapping out our attack surface we began by first narrowing done the codebase to our target.

First off, the repository contains the `code` directory which containers all of our base code. In this repoistory we find a README.md that lays out which direcotries contain which code. Since our target is the FiveM client we can target most of the code in `/client` however there will be shared code called from libraries so we will be jumping between mostly `components`, `deplibs`, `shared`, and `server`.

Looking closer at `client` we can see it has a similar README.md file to map out the codebase. In here we have a few areas of interest in the `citicore` directory including an in game console where we can run commands and *very limited* code. This ended up being one of the directories we began reading in more closely.

In addition to client, we looked closely at the `components` directory which included a lot of the helper functions scipts use to call in-game actions. These in-game actions are hooks to C/C++ code which then passes the arguments to functions in the game engine. This is vital code for the game to work and also very

prone to errors and bugs which we will look into. The onyl problem with this direcotry is that it contains the code for both the FiveM and RedM (a modification for another Rockstar game). However the README.md file specifies which libraries are for the server, FiveM, or shared libraries.

Plans For the Rest of the Project

After mapping out the codebase and figuring out what code does what, we began to read code across the codebase as well as reference this blog post we found about exploiting FiveM in the past. However before we can go into future plans, let's highlight what we looked at so far and our findings.

What We Found So Far

Working off the blog post we found the `InvokeNative` function they mentioned and began to statically analyze it to see if we could find a bug in the up-to-date version of it.

```
class NativeInvoke
{
public:
    template<uint64_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);
        }

        cxt.SetVectorResults();

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

NativeInvoke is a helper function used to call the game engine so developers can script out certain behavior in game (like decremented health after being damaged). We can see here it takes a hash address as well as arguments which it pushes into a `NativeContext` class. Looking at this, we found a possible vulnerability:

```
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;

        memset(m_TempStack, 0, sizeof(m_TempStack));
    }

    template <typename T>
    inline void Push(T value)
    {
        if (sizeof(T) > ArgSize)
        {
            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` function we called earlier first does some size checks to make sure our args are valid before throwing it onto a stack. However, importantly it does not check if the amount of args exceeds the max args. This points to a likely buffer overflow attack where we pass in our payload 8 bytes as "args" until we overflow the return address and then begin a ROPchain.

Future Plans

To expand on this we'll likely test this by replicating this bug by calling `NativeInvoker` via a lua script and seeing if a buffer overflow is possible. If we find this is not exploitable we will look at similar functions in the `components/rage-scriptng-five` directory as it seems to have a lot of potentially unsafe low level code.

We also plan to do a bit of code review in the `/client/citicore/console/CoreConsole.cpp` file as it allows users to run *limited* code in their FiveM console and has a lot of strange regex checks that we would like to look into.

If we do not get a possible exploit after sufficient code review, we plan to go either two routes: static or dynamic analysis. Static analysis would have us writing simple scripts to search the codebase for unsafe functions and look for references to said unsafe functions. With dynamic analysis, we plan to set up a simple fuzzer against places that take user input (like the console and lua scripts). Against the console, it will likely be sufficient to do a normal black box fuzz as it will pass our input directly to the C++ code (possibly causing bugs). However, if we were to exploit the Lua library hooks we would need to set up a grammar based fuzzer and define a simple grammar for Lua (types and variables not really functions) and the functions we would like to fuzz (namely functions like NativeInvoke).

As stated earlier however, our first priority is exploring the possible buffer overflow we found in the `NativeInvoke` function and seeing if it is possible to crash and/or exploit with it.