Gamehacking in Rust

Eigenvektor

August 23, 2018

This document tries to give an introduction to gamehacking with Rust. It focuses on the Windows side of things, with Linux coming soon™

1 Necessary Tools

1.1 Rust

To setup Rust, all you need to do is to download the installer and execute it. You might want to use the nightly toolchain. Also you will need MSVC.

1.2 Editor

I recommend VSCode. You will need the Rust (rls) Addon, also I will recommend you to install the crates Addon to check if your crates are up to date.

2 Getting started

2.1 Creating and configuring a new Project

The first step should be creating a new Rustproject using "cargo new project_name". This will create a new folder with all the needed files. After that try you should configure your cargo.toml, which is the build file for Rust. Add the following text to your cargo.toml to instruct cargo to build a dll:

```
[lib]
name = "binary_name"
crate-type = ["dylib"]
```

Also I recommend you to install the following crates:

```
[dependencies]
winapi = "0.3.5"
detour = "0.5.0"
chiter = "0.3.0"
```

Keep in mind to adapt the version to your needs. Another nice to have crate is libc for easier FFI.

2.2 Getting started with the code

With version 0.4.0 of chiter, which is not released at the time of me writing this, will include a macro to create an entrypoint for the dll easily, all you have to do is:

```
#![feature(use_extern_macros)]
extern crate chiter;

//To access the Windows-API
extern crate winapi;

use std::thread;

fn entry_point() {
        //Your code here
}

//This creates the DllMain and a new thread which will run entry_point
chiter::make_entrypoint!(entry_point);
```

If you are unable to use chiter, here is the macro source:

```
macro_rules! make_entrypoint {
        ($fn:expr) => {
                #[no_mangle]
                pub extern "stdcall" fn DllMain(
                _hinst_dll: winapi::shared::minwindef::HINSTANCE,
                fdw_reason: u32,
                _: *mut winapi::ctypes::c_void,
                )        {
                        if fdw_reason == 1 {
                                thread::spawn($fn);
                        }
                }
        };
}
```

If you want to test it, you should modify your cargo.toml to use the winapi-feature consoleapi. This is done by replacing the current winapi line with

```
winapi = {version = "0.3.5", features = ["consoleapi"]}
```

Then enter the following code into your entry_point function:

```
unsafe {
        winapi::um::consoleapi::AllocConsole();
}
println!("Injected!");
```

2.3 Hooking

To easily hook functions, I recommend the detours crate. Just include it using

```
extern crate detour;
//GenericDetour allows infinite Hooks, StaticDetour only 1
use detour::{GenericDetour, StaticDetour};
```

I will now show an example to create an GenericDetour:

```
//In my case DoEmote has the following c prototype: static bool __stdcall
//DoEmote( unsigned char id );
static mut DO_EMOTE: Option<GenericDetour<extern "stdcall" fn(u8) -> bool>> = None;

const DO_EMOTE_OFFSET: u32 = 0xDEADBEEF;

//This will the function to be executed after the hook is placed
extern "stdcall" fn hooked_emote(data: u8) -> bool {
        true
}

//This this the function that will be called instead of the original function
extern "stdcall" fn hooked_emote(data: u8) -> bool {
        println!(
                "[EMOTE] ->Ping type is {}",
                data
        );
        unsafe {
                match DO_EMOTE {
                        Some(ref mut hook) => hook.call(data),
                        None => {println!("Could not invoke hook"); false},
                }
        }
}

fn entry_point() -> bool{
        unsafe {
                winapi::um::consoleapi::AllocConsole();
        }
        println!("Injected!");
```

```
        //Make a function pointer
        let orig_do_emote = unsafe {  std::mem::transmute::
                <*const usize, extern "stdcall" fn(u8) -> bool>(DO_EMOTE_OFFSET as
                 *const usize)
        };

        unsafe {
        //This will place the hook, but it will be inactive
                DO_EMOTE = Some(
                        GenericDetour::<extern "stdcall" fn(u8) -> bool>
                        ::new(orig_do_emote, hooked_emote).unwrap(),
                );
        }

        //Enable Hook
        unsafe {
                match DO_EMOTE {
                        Some(ref mut hook) => {hook.enable().unwrap();
                        println!("Hooked DoEmote");},
                        None => println!("Could not enable execute hook"),
                }
        }

        unsafe {
        //This will call the original function, all calls to
        //orig_do_emote will now result in hooked_emote being called
                match DO_EMOTE {
                        Some(ref mut hook) => hook.call(data),
                        None => {println!("Could not invoke hook"); false},
                }
        }
}
```

Note that I am not that good in Rust, so this Code might be pretty bad, but it works^{TM}.
An example for a StaticDetour can be found here.