GUIDE TO LUA DETOURING APPLIED TO GMOD

Since gmod is slowly dying, I decided to release some public documentation to help hackers interested in advanced lua sorcery.

The purpose of this guide is to share knowledge and explain the theory behind detouring functions in order to bypass/make anticheats on Garry's Mod. I'll only talk about CLIENTSIDE detours. (more focused on how to cheat rather than making anticheats)

1/ What does detouring functions mean?

When you detour a function, you're simply overwriting it with your own custom function to make it do whatever you want. Detours only apply to global functions, as there's no point in overwriting your own local functions that only you can access.

Here's a really basic example of what a detour looks like:

```lua
function PrintTable(tbl)
    return print("Hello World")
end
```

This code overwrites the function PrintTable and makes it print “Hello World”.

```
] lua_run_cl PrintTable({})
Hello World
```

In this case, the detour completely changes the behavior of the original function. If you want to a function to still work as intended while being detoured, you have to locally copy the function before detouring it, and then you simply return the copy in the detour. Here's an example:

```lua
local Original_PrintTable = PrintTable

function PrintTable(tbl)
    return Original_PrintTable(tbl)
end
```

Now the function still works while being detoured:

```
] lua_run_cl PrintTable({1,2,3})
1       =       1
2       =       2
3       =       3
```

You can add anything in the function before returning the original copy.

2/ Why is detouring overpowered?

When it comes to cheating on garry's mod (or detecting cheaters), whoever loads first just wins the fight.
Indeed, all the clientside files have access to the same global clientside functions. If a file loads and detours PrintTable for example, all the other files loading afterwards will only be able to access the detoured version of PrintTable. The first file to detour PrintTable is the only file that will be able to access the original undetoured version of that function (if it made a local copy of it before detouring it of course).
Clientside anticheats usually use a bunch of functions to “scan” the clients, hence by loading before the anticheat you gain the ability to spoof anything and control what the functions return with your detours.

Thus, cheating with lua on garry's mod without being detected relies on only two things:

- loading as early as possible
- detouring a bunch of functions

The former being easier to do than the latter, for anyone having a bit of c++ knowledge.

If you don't know how to load your lua before autorun, you should first look into that before going further in the guide. There is some documentation on how to do it, here's a hint: https://pgl.yoyo.org/luai/i/luaL_loadbuffer
Note that you need to make sure you load before any anticheat. The earliest loading spot of anticheats being the top of the init.lua file, loading your cheat before init.lua is the best way to do it.

3/ What functions should you detour?

Once you've got your working preinit module allowing you to load your cheat before autorun, you'll probably ask yourself this question. What the hell are you supposed to detour? Well there's a TON of functions you need to detour. It really depends on the anticheat you want to bypass to be honest, but the goal for you should be to be as hard to detect as possible, not to just bypass one shitty anticheat. It wouldn't make any sense to just fix one flaw in your cheat and leave it 95% detectable instead of 99%.
So, the functions you need to detour are the functions anticheats can possibly use against you to notice something is wrong with your client. The most important ones to detour are:

- the whole debug library
- the whole jit library
- gcinfo & collectgarbage (used to retrieve lua memory usage)
- tostring & string.format (used to retrieve functions addresses)
- the hook library (if your cheat uses hooks, you need to hide them)
- the concommand library (if your cheat uses concommands, you need to hide them)

4/ debug.getinfo

debug.getinfo is probably the most important function to detour in the debug library.
Indeed, this function can be used to retrieve a table with a bunch of information about whatever function you passed as the first argument. (you can learn more here https://wiki.facepunch.com/gmod/debug.getinfo)
The function allows you to know where a function was defined, in which file, at what lines, if it's a C or a LUA function, etc...

Anticheats generally use debug.getinfo to check if a function has been detoured or not.

Here's an example of what the information of the original debug.getinfo function look like:

```
] lua_run_cl PrintTable(debug.getinfo(debug.getinfo))
currentline     =       -1
func    =       function: builtin#125
isvararg        =       true
lastlinedefined =       -1
linedefined     =       -1
namewhat        =
nparams =       0
nups    =       0
short_src       =       [C]
source  =       =[C]
what    =       C
```

Now if we detoured debug.getinfo like so:

```lua
local Original_Debug_Getinfo = debug.getinfo

function debug.getinfo(...)
    return Original_Debug_Getinfo(...)
end
```

Here is what the information would be:

```
] lua_run_cl PrintTable(debug.getinfo(debug.getinfo))
currentline     =       -1
func    =       function: 0x50a1fe60
isvararg        =       true
lastlinedefined =       7
linedefined     =       5
namewhat        =
nparams =       0
nups    =       1
short_src       =       lua/cheat.lua
source  =       @lua/cheat.lua
what    =       Lua
```

Any anticheat could detect that. So before detouring anything, to make your detours are undetectable, you need to detour debug.getinfo, so anticheats can't see the difference between a detour and the original function.
Make sure you always store the original version of a function before detouring it (store local copies in a table).
In your debug.getinfo detour, you need to:

- check if the function being passed if one of your detours
- return the information of the original copy instead of the information of the detour

To store functions, I'd advice you to copy _G (which is the global table containing all the global functions).
At the top of your cheat, just add that line:

```lua
local G = table_copy(_G)
```

Note that _G does not have ALL the functions before init.lua loads.
A lot of functions (LUA functions, not C) get progressively added as init.lua loads different extensions/modules.
Obviously, you can't detour/copy/use a function that doesn't exist. If you need to use a function before init.lua imports it, you need to import it yourself (which is easy since the functions are made in lua, you just click the view source button on the wiki and paste the code in your cheat, don't forget to localize it tho.)
table.Copy being a lua function that gets added when the extension table.lua loads, you need to import it so you can copy _G right when your cheat loaded.

Once you've copied _G, you can access all the original copied functions without manually having to copy them, which will make your detours quicker to do:

```lua
local G = table_copy(_G)

function debug.getinfo(...)
    return G.debug.getinfo(...)
end
```

The next thing I recommend making is a function you'll call everytime you detour a function. The function will store the detours and original functions in a table. The detours will be the keys and the originals will be the values:

```lua
local Detours_To_Originals = {}

local function StoreDetour(original, detour, name)
    if not original then print("attempt to detour non existing function") return end
    Detours_To_Originals[detour] = original
    print("Successfully detoured "..name)
end
```

Now in your debug.getinfo detour, you can add a simple condition to check if the function being passed is a detour or not, and return the information of the original function, like so:

```lua
function debug.getinfo(func_or_level, fields)
    if Detours_To_Originals[func_or_level] then -- if the func is one of your detours
        return G.debug.getinfo(Detours_To_Originals[func_or_level], fields) -- return the infos of the original func
    end
    return G.debug.getinfo(func_or_level, fields)
end
StoreDetour(G.debug.getinfo, debug.getinfo, "debug.getinfo") -- we need to call that after every detour
```

Congratulations, you have a decent debug.getinfo detour hiding the information of your detours. Now, everytime an anticheat will attempt to get the information of a function by using debug.getinfo, it'll call your detour, and the detour will spoof the info if necessary. The same logic applies to a lot of detours. This debug.getinfo detour remains severely incomplete, there's a lot of little things to handle to get it to work perfectly, I might show some later.

Note: If you read the wiki page carefully, you probably noticed that you can also pass stacklevels instead of functions in the first argument of debug.getinfo. So you need to make your detour handle stacklevels too. The way you deal with stacklevels to avoid being detected is WAY more complex. I'll talk about stacklevels in the next section.

5/ Stacklevels

What is a stacklevel and why is it important?

https://www.lua.org/pil/23.html:
“An important concept in the debug library is the *stack level*. A stack level is a number that refers to a particular function that is active at that moment, that is, it has been called and has not returned yet. The function calling the debug library has level 1, the function that called it has level 2, and so on.“

There's a few different ways of dumping/accessing stacklevels:

- passing a number instead of a function as the first argument in debug.getinfo
- using debug.traceback (dumps the whole stack)
- calling getfenv on a number (doesn't really get the stacklevel, but gets the environment of the function at that level)
- throwing an error (yeah this one is kinda op)
- debug.getlocal

An anticheat can dump the stack at any moment and check for suspicious things/foreign sources inside.
This is why you should find a way to deal with stacklevels in your debug.getinfo detour. Keep in mind you need to make sure getfenv, debug.getinfo and debug.traceback return the same information concerning stacklevels.

You can use different methods to deal with stacklevels:

- everytime your cheat's functions get found at one level, just spoof it with some random legit function
- or instead of spoofing things you could simply completely remove that level from the stack (way harder but less detectable)

The second method is way better and less detectable, but you need some really advanced detours to do it properly, and a lot of knowledge. Altho I won't spoonfeed you how to remove things properly from the stack, I'll show you a basic example of how you spoof them:

```
function debug.getinfo(func_or_level, fields)
    // treating numbers (stacklevels)
    if G.type(func_or_level) == "number" then
        local infos = G.debug.getinfo(func_or_level, fields) -- getting the infos table at that level
        if infos.short_src and infos.short_src == "CHEAT_SOURCE" then -- checking if its our cheat
            return G.debug.getinfo( G.pcall ) -- spoofing it
        end
    end
    return G.debug.getinfo(func_or_level, fields)
end
StoreDetour(G.debug.getinfo, debug.getinfo, "debug.getinfo") -- we need to call that after every detour
```

Here everytime your cheat's functions leak in the stack it'll replace the info with the information of the function pcall. (you can spoof it with any other legit func)

6/ jit.util.funcinfo

jit.util.funcinfo globally works like debug.getinfo does, it returns a table of information concerning a function. Hence, it is as important as debug.getinfo and you need to detour it too. (the detour is the same as debug.getinfo, you simply check if the argument passed is a detour and spoof the return into the original func)

7/ jit.attach

jit.attach is another really important function you need to detour. The detour itself is quite complex.
First, you need to understand how the function works and what it does, which you can do by simply looking it up on the wiki. (https://wiki.facepunch.com/gmod/jit.attach)
The function allows you to attach a function to a jit compiler event. You can find the list of all the events on the wiki.
Let's take the example of the “bc” event:
It is triggered everytime a function gets compiled to bytecodes. The function being compiled is passed as the first argument, in the form of a proto function. (there is almost no documentation on them, but a proto is a function in it's early stage before it gets compiled into bytecodes, you can't call debug.getinfo on a proto, you must use jit.util.funcinfo to get its information)

Every single lua function being executed needs to get compiled first, which means this jit.attach event catches EVERYTHING.
Anticheats will use that and attach a function that checks every single compiled function.
Note: You can attach a function using jit.attach, but you can also manually attach it by adding it in the registry _VMEVENTS table. (whenever jit.attach attaches a function, it just puts the function inside that table with the hashed event as the key and the function as the value). Thus detouring jit.attach won't be enough, you also need to find a way to takeover the control of that _VMEVENTS table if you want to remain undetected.

8/ tostring/string.format

When you overwrite a function with your detour, it changes the function's address. A lot of C++ functions have a static address (like debug.getinfo for example), which allows anticheats to check that to see if it changed. The only two functions used to get an object's address are tostring and string.format. To avoid being detected, you need to detour them and return the original address so your detours stay undetected. Concerning tostring, you just apply the same logic: add a condition to check if the object is one of your detours, and spoof the return if its the case.
Concerning string.format, the detour is a bit more complicated since the second argument is a vararg (meaning you can pass multiple objects and they'll all get formatted).

9/ gcinfo/collectgarbage

The next concept you need to be aware of when messing with your cheat to bypass anticheats is the lua memory usage. gcinfo/collectgarbage are the only two functions you can use to see the lua memory usage in real time.
Those functions return numbers that correspond to how much memory lua is actively using.
Note: lua memory works like a cycle: the memory increases over and over until it gets collected and returns to 0. (sometimes it even goes negative for a frame)
So the more things are happening on your client (the bigger your cheat is) the higher the lua memory will be.
Anticheats generally dump the client's memory usage before autorun (really early in init.lua). Because they do it early, the memory usage is supposed to be low. But since your cheat loaded before the anticheat and already processed a lot of things, your memory will be higher, exposing you to some detection. You need to fix that by detouring the two functions and substracting the additional memory your cheat takes.

10/ Hiding hooks

First, let's talk about how the hook library really works. If you look at the source in hook.lua, hook.Add only adds a bunch of information (names, events, functions) in the hooktable.
The functions running the hooks are hook.Run and hook.Call.
Hence the simplest way to make your hooks not appear in the hooktable is to just add your cheat's hooks in a separate local table (make sure the table is formatted like the normal hooktable). Once you've got your local cheathooks table, you just detour hook.Call and loop through your cheathooks table to run them all (run the functions using pcall for more safety). If you do everything correctly your hooks will successfully run without showing up in the hooktable. (no need to detour hook.GetTable)
Note: in some special cases, if the server you're playing on has some specific addons overwriting the hook.lua module, you'll need to re-detour the functions. (Ulib does that for example)

11/ Hiding concommands

You can apply the same logic for the concommand module. The function concommand.Add stores concommands and functions in a table to access and run them later. Simply add your cheat's concommands in a local table and detour concommand.Run to make it run your concommands as well as the legit ones.

12/ Sandboxing your cheat

The next concept you need to understand in order to write an undetectable good lua cheat is sandboxing. What does it mean? Basically, sandboxing your cheat means loading it in a safe environment containing only original local undetoured copies of functions. An example of a safe environment is the table you created earlier to return the copies in your detours. You need to make sure your environment only has safe functions, so you need to create the table and copy functions into it really early, before any addon/anticheat could have loaded. (basically as soon as you load)

But as I told you earlier, if you use table.Copy to copy all functions in _G in preinit, your safe environment won't have all the functions since a few modules/files haven't loaded yet. (you won't have hook functions for instance...) So what you need to do is find a way to know when exactly those modules and files load, and copy the functions they added as

soon as possible. (basically updating your safe env) The best way of doing that is to detour the functions used to load files and to just update your environment in there. (include/require)

Once you have that environment, you can now load anything in it. (if you did something wrong or forgot to add a few functions in your environment, it'll error since you'll be trying to execute a function that doesn't exist in it)

Note that there is two different ways of "sandboxing" something. Whenever you call a global function without specifying the environment, it'll just find and execute it from the default global environment which is _G. (print and _G.print are the same functions) That means that you can "manually" sandbox your cheat by just manually specifying the environment of the functions when calling them in your code. Basically, you never want to use print, because you would be calling _G.print which is detectable. Instead, just manually call G.print (if G is your safe env). The only drawback of this method is that it's extremely annoying and unpractical, most people are just used to using default functions and will easily forget to call them from their safe env manually.
The second method is to just use debug.setfenv (https://wiki.facepunch.com/gmod/debug.setfenv) to force your function/code to execute in a specific environment. Thus you can just code and call the global functions normally, but instead of being called from _G, they will be called from your safe environment. Here is an example of how you use debug.setfenv to achieve this:

```lua
local G = table_Copy(_G) -- safe env

local function LoadCheat()
    -- hacks!!!
end

G.debug.setfenv(LoadCheat, G)

LoadCheat()
```

Note that the global environment table contains itself, meaning you can do: print(_G._G._G._G._G).
So you need to make it so your safe env contains itself at that _G key like so:

```lua
local safe_env = table_Copy(_G)

safe_env._G = safe_env
```

13/ Conclusion

To sum it up, here are the general steps for making sure your cheat is almost undetectable:

- load before everything else
- copy all the original functions (create your safe env)
- detour all the functions any anticheat could use against you
- sandbox and load your cheat in a safe environment

There you go, have fun.

14/ Credits to a lot of friends

If you're on this list, you're a really good friend who helped me with a lot of stuff over the years:

- Badger
- bsn/t0ny
- banksy
- THE GRUMP
- Ted

If you want to contact me, you can add me on steam (https://steamcommunity.com/id/lIIIIIIlllIIIIIIIIIIl/) or discord (wayblund#6714).