BLT Modding

Introduction

This guide will be a simple tutorial on how to make your first BLT mod with a suitable example to follow along.

The example mod will hide all (multiplayer) jobs from Crime.Net which are not stealthable.

Scope

It it recommended to have at least beginner level knowledge in programming (doesn't have to be Lua necessarily). These terms shouldn't be new to you: *Variables, Conditions, Functions*

Expect this:

- An introduction into BLT-modding
- Understanding core concepts
- A working example to play around with
- Links to resources

DON'T expect this:

- Learning how to code Lua
- Understanding the Payday 2 source code
- Using libraries like BeardLib

Setup

- Make sure you have a code editor. I can recommend <u>Visual Studio Code</u>

- Download the <u>Payday 2 Source Code</u> (only used for searching - you don't need to build anything)
- Enable the developer console by placing an empty `developer.txt` file inside your `mods` folder. (This can help you find errors in your mod)
- Make a new folder in `mods` and place an empty `mod.txt` and `mod.lua` inside it.

Beginner BLT modding

mod.txt

Every mod needs this file. This file is responsible for loading your mod with BLT. Below you will find the `mod.txt` used for this example. Copy the contents inside your own `mod.txt`. Most of the fields should be self explanatory expect for the `hooks`.

```
{
    "name": "Hide loud jobs",
    "description": "Hides jobs from crime.net which are not stealthable",
    "author": "Your name",
    "version": "1.0",
    "blt_version": 2,
    "hooks": [
        {
            "hook_id": "lib/managers/crimenetmanager",
            "script_path": "mod.lua"
        }
    ]
}
```

Hooks

- `hook_id` is the path to the file (from the original codebase) in which the function is defined which we want to manipulate.
- `script_path` is the path to the file in your mods folder where your code is in.

Finding the code

There's no magic trick to finding the right place in the source code. It takes practice and understanding of the codebase. We will not be covering that here but you are more than welcome to ask the community for help in the `#payday-help`-channel of our <u>Discord</u> on this topic.

You can still follow along by opening the file `lib/managers/crimenetmanager` (this is also the `hook_id`) inside the source code you downloaded earlier. Find the function `CrimeNetGui:add_server_job` (or view it [here](#)).

Editing the code

Copy the entire function in your `mod.lua`. Now we will add an `if`-statement around the code inside the function. This statement will check if it is indeed a stealthable mission. Only if that's the case will the game actually add it to the map. Luckily there is already such a function built into the game (as it's used to display the little ghost icon next to the job). You can find that part in the same file (or view it [here](#)). Let's have a look at it:

```
if data.job_id and managers.job:is_job_ghostable(data.job_id) then
 ghost_icon = icon_panel:bitmap({
  texture = "guis/textures/pd2/cn_minighost",
  name = "ghost_icon",
  blend_mode = "add",
  color = tweak_data.screen_colors.ghost_color
 })

 ghost_icon:set_top(side_icons_top)
 ghost_icon:set_right(next_icon_right)

 next_icon_right = next_icon_right - 12
end
```

From simply reading the code: If the `job_id` exists and the job with the given `job_id` is stealthable, we create a new icon with the texture of the blue ghost and set its position. Now let's try to apply that to our code. Our function is also given the parameter `data`. This is in fact the same object for this case. Therefore we can safely wrap our code in the exact same `if`-statement like this:

mod.lua

Here's how your finished mod.lua should look like:

```
function CrimeNetGui:add_server_job(data)
 if data.job_id and managers.job:is_job_ghostable(data.job_id) then
  local gui_data = self:_create_job_gui(data, "server")
```

```
  gui_data.server = true
  gui_data.host_name = data.host_name
  self._jobs[data.id] = gui_data
 end
end
```

That's it - Try it out!

► End of basic tutorial ◄

Continuation: Finding the code

I will explain to you my thought process of how I was able to find the correct function in case it may help you:

For this example I was specifically looking for `crimenet` which brought me directly to the `CrimeNetManager`. I knew that stealth is mostly referred to as `ghost` in the code - so I quickly picked up on the idea of the `ghost_icon` which was used in the function `_create_job_gui` but that one is difficult to grasp and gets called through the more accessible function `add_server_job`. Now I only had to go back to the `ghost_icon` to figure out how the game determined if it should be displayed or not.

Different methods:

Let's review some other methods we could have used to get this mod working.

[Hard Overwrite] (as shown above)

- PRO: You are in full control by modifying the original source code.
- CON: Only 1 mod installed can hard-overwrite the function (not necessarily yours). All other mods manipulating the same function will not work.

```
function CrimeNetGui:add_server_job(data)
 if data.job_id and managers.job:is_job_ghostable(data.job_id) then
  local gui_data = self:_create_job_gui(data, "server")
```

```
   gui_data.server = true
   gui_data.host_name = data.host_name
   self._jobs[data.id] = gui_data
  end
 end
```

Explanation:

This method is the most straightforward: It is the exact copy from the source code but with our adjustments.

[Soft Overwrite]

- PRO: good for functions which return values.
- CON: Still issues which compatibility as functions are chained/nested inside each other and depending on loading order other mods could alter that behavior.

```
local old_add_server_job = CrimeNetGui.add_server_job

function CrimeNetGui:add_server_job(data)
 if data.job_id and managers.job:is_job_ghostable(data.job_id) then
  old_add_server_job(self, data)
 end
end
```

Explanation:

This method works by keeping a copy to the original function. This function is then overwritten with our implementation where we call the reference to the original function.

[Hook]

- PRO: The original function stays completely untouched.
- CON: Does not work for functions which return values. We will cover techniques to circumvent these issues in the "Advanced BLT modding".

```
Hooks:PostHook(CrimeNetGui, 'add_server_job', 'hideloudjobs_crimenetgui',
function(self, data)
 if data.job_id and not managers.job:is_job_ghostable(data.job_id) then
  self:remove_job(data.id, true)
 end
```

```
end)
```

Explanation:

Hooks don't touch the original function. They get executed just **before** (Pre) or **after** (Post) the function.

So which one should you use?

Generally speaking: Hooks > Soft Overwrite > Hard Overwrite

Use Hooks wherever you can. Use Soft Overwrites when dealing with function where you need to change the returning value and only use the Hard Overwrite method if you really have to.

Resources

- [Decompiled game code](#)
  Source code *of the game*

- [Original BLT](#) (*outdated on some occasions*)
  Documentation *which holds information about many features of (Super-)BLT like Menus, Localization, Networking ...*

- [Community BLT](#) (*work in progress*)
  Documentation *which holds information about payday 2 mod creation, including examples*

- [BeardLib](#)
  Library *for adding custom content, making menus ...*

- [Auto Menu Builder](#)
  Library *for generating menus*

- [GitHub Auto-Update](#)
  Tool *for making automatic updates via GitHub*

- [Resources & Libraries](#)
  Link *to the category from the MWS site*

Quick introduction to Hooks

This guide is intended for people new to PAYDAY 2 Lua modding and explains different methods of hooking functions and when to use them. I'll go over the most commonly used methods of function hooking and explain how they work and why we need them.

Why bother using hooks?

You may ask yourself: Why all this complicated stuff? Can't I just copy and paste the original function and make my edits to it? While yes, you could do a complete *function override*, you really shouldn't unless it is absolutely unavoidable. In the worst case this will crash the game when an update changes some things in that function and you will have to copy and change the new function again to fix it. You will also run into the risk of breaking mods that rely on the same function, as your mod would completely undo changes of other mods if it loaded afterwards. The major takeaway is therefore: Doing proper function hooks increases compatibility with other mods and game updates and is less likely to break over time. With that out of the way, let's see what kind of hooking techniques exist.

Types of hooks

PostHook

Depending on what your mod is trying to achieve, you will have different needs when hooking functions. The most common scenario is probably adding additional data or code on top of an existing function after it executes. For this we generally use SuperBLT's *PostHook*, which registers a custom function that is executed whenever the game executes the target function. Using a *PostHook* is simple and can be used if you don't need to change the functionality of the original function and just want to add some additional code.

Let's look at an example: We want to change the HP of the regular street cops. We can do this after the cop's stats have been initialized, which happens in the function `CharacterTweakData:_init_cop`. Looking at this function in the game code we can see that the cop's health is set here. Since we just want to change the HP and still want all the other things to be initialized like usual we can add code in the form of a *PostHook*.
We would do that like this:

```
Hooks:PostHook(CharacterTweakData, "_init_cop", "our_unique_hook_id", function (self)
	self.cop.HEALTH_INIT = 10
end)
```

When the game runs the original `CharacterTweakData:_init_cop` function, all the cop data will be initialized as usual, but immediately afterwards, BLT will execute our hook function, changing the cop's health to the value we supplied.

The function you specify for the *PostHook* will be supplied with the same arguments that the original function will be called, so if you need them, you can specify them.
Note that `CharacterTweakData:_init_cop(presets)` is functionally identical to `CharacterTweakData._init_cop(self, presets)` (specifically note `.` instead of `:` between `CharacterTweakData` and `_init_cop`). This is important for your hook function if you plan to make use of the function arguments and the reason why you usually see `self` as the first argument in a hook function even if the original function doesn't have it.
If you are unsure about wether to include `self` in the list of arguments for your hook, just remember that if the function is defined with a `:` in the game code, the first argument to it in a hook should be `self`.

If you need access to what the original function of your hook returned you can use `Hooks:GetReturn()` which will return any values that have been returned by hook functions and the original function call that ran before your hook function. Let's say you made a new custom enemy and for making it work properly you need to add it to the character map (a list of all enemies the game goes over and generates contour mappings for). The character map is returned by the function `CharacterTweakData:character_map` and the function itself creates and returns a local table with all characters in it.

```
Hooks:PostHook(CharacterTweakData, "character_map", "our_other_unique_hook_id", function (self)
	local char_map = Hooks:GetReturn()
	table.insert(char_map.basic.list, "custom_enemy_name")
	return char_map
end)
```

You can see that it easy to access the return value and change it (in this case the return value is a table and we insert an entry into it. Another thing that you should notice is that you can return values from a hook, this will override the return value of the original function (and any function hooks on the same function that came before yours).

PreHook

Very similar to a *PostHook*, the only difference is that it will be executed before the original function is called. Less commonly used but useful if you need to change some values or add additional code right before a function call. Note that setting fields that are created or set in the

function in the original code will have no effect since your code runs before the original function and the original call will just override any values that it sets.

Another niche use case for a *PreHook* is changing function arguments that are of table type, as tables are passed by reference and you can therefore change the content of the table which will then be passed to the original function. As other data types are passed by value this only works for tables.

Function wrapping

The above mentioned hooking methods should cover a lot of usecases already, but there are some cases where they are not usable. Let's say you want to change the arguments the original function is called with or stop the function from being called based on some condition. This simply can not be done with a *PostHook* or *PreHook* since all function arguments except tables are passed by value and *PostHook* or *PreHook* will always run the original function.
In this case, you need to do a *function wrap*, often referred to as *old_init*. This involves saving the original function into a variable and then overriding the function with a new one. In the new function you then call the original function manually and do whatever else you need to do.

```
local build_suppression_original = CopDamage.build_suppression
function CopDamage:build_suppression(amount, ...)
	if amount == "max" then
		amount = 2
	end


	return build_suppression_original(self, amount, ...)
end
```

Some notes on this code:

- Using ⬚…⬚ in the function arguments represents any number of additional unspecified arguments. We can use this to make sure we pass every argument that our function is called with to the original function call without actually caring what they are. If you need some of the actual arguments, you can simply list all arguments up to the ones you need and then follow them by ⬚…⬚. It is suggested to always use ⬚…⬚ at the end of the argument list, even if you already listed all of the original arguments in case any other mod or game update adds additional arguments to that function call.
- We have to make sure to return whatever the original function is expected to return, as returning a wrong type or nothing at all can lead to crashes that are hard to pin down.

So you chose to override after all

If none of the methods above can be applied to what you want to do, you will have to override the entire function. However, SuperBLT provides a way to do this in a way that at least keeps any hooks made by other mods intact.
An example could be the following:

```
Hooks:OverrideFunction(GroupAIStateBesiege, "assign_enemy_to_group_ai", function (self)
	-- Your code here
end)
```

Note that if another mod used SuperBLTs *PostHook* or *PreHook* on the same function, they will still be called and you can maintain *some* compatibility with other mods. Redefining the function without making use of `Hooks:OverrideFunction` will remove all hooks that were made by mods that ran before yours which could lead to unexpected behavior.