Did you ever wonder what keeps you from developing an addon that completely automates your game? Well, such an addon would break the EULA, and you don’t want to do that because you don’t want to lose your World of Warcraft account. But what keeps you technically from developing such an addon? The whole UI is written in Lua and XML, so there are functions that allow you to move your character and to cast spells. For example, there is the function CastSpellByName(“name”), which allows you to cast a spell. But try to do this with a simple spell like Attack:

You will get an error message saying that your action has been blocked because it is only available to the Blizzard UI. In this chapter we will explore how the game determines what belongs to the default UI and what does not. We will also discuss the circumstances under which we are able to cast spells and execute certain restricted actions from an addon.

Secure and Tainted Code

All Lua values and references are always either secure or tainted in World of Warcraft. Everything that is created from an addon or via slash commands will be tainted, and everything that is created by the game is secure. The current execution path (an execution path is the currently running code, which is started by the game and ends as soon as all called functions have returned) can also be either secure or tainted, but it always starts as secure. It will become tainted as soon as it encounters anything that is tainted.

The game uses digital signatures to check the integrity of the default UI when it is loaded, and only code that has been digitally signed by Blizzard will be flagged as secure upon loading, so every value created by the Blizzard UI is flagged as secure. This also means that all of your addons are automatically tainted and all values you create are also tainted.

What does this mean for you? There are certain functions, known as restricted functions, that can only be called from a nontainted execution path. One of these restricted functions is CastSpellByName. But we don’t need to discuss all of these functions here, as we will never be able to call any of them. Let’s see an example demonstrating tainted and secure code.

Restricted Function

Before the taint system was introduced in patch 2.0, I once wrote an addon that prevented certain spell casts in certain situations. Why would I want to do this? The addon was a boss mod for Nefarian in Blackwing Lair. Just in case you don’t know, this really old boss, Nefarian, places a debuff on a certain class from time to time that corrupts some of the class’s spells and abilities. For holy priests like me, this debuff turns all healing spells into damaging spells, so it is a really bad idea to continue healing the tank when you’ve got this debuff. The addon detected this and blocked all healing spells while the debuff was active.

Blocking spells worked with hooks in a few functions in this addon; one of these hooks was in the function CastSpellByName. Let’s try to write a simple hook for this function. Note that this function is invoked only when you use the /cast slash command (for example in a macro) but not when you click a button in your action bar. Action bars work slightly differently (we will see them later in this chapter), and I will only show you the hook into CastSpellByName to keep it simpl.

The following code can easily be executed in the game with an addon like TinyPad:

This should print all arguments passed to CastSpellByName and then call it. The actual addon had a simple check for a variable that was set if the debuff was active. Now try to execute the following slash command or to use a macro that casts a spell

You will get the same error message as if you had called CastSpellByName directly, even though the actual call to it comes from the default UI. Our simple hook just broke all slash commands that can cast spells; /castrandom and /castsequence are also affected

Let’s see what happened by tracing the call. The execution path starts in the DnEnterPressed event of your chat frame as a secure execution. It then goes through a lot of code in the file FrameXML\ChatFrame.lua that identifies your command as a slash command until it finally reaches its slash command handler in the table SecureCmdList. This works just like normal slash command handlers; don’t worry about the different name. The handler parses the slash command, determines whether you are trying to use an item or a spell, and gets the optional target argument. The execution is still secure when Lua reaches the following handler, which is also defined in the file FrameXML\ChatFrame.lua:

It now calls the function stored in the global variable CastSpellByName, which is normally secure. But we overwrote it with our custom hook function, which is not secure

The execution path now becomes tainted the moment it enters our function. This tainted execution now calls the real function CastSpellByName, which is still secure (but the reference stored in the local variable old is tainted; you will see how exactly the spreading of taint works in just a bit). It doesn’t help us that the function is still secure—it is called from a tainted execution path and therefore fails. The only way to get rid of this taint is to reload your user interface

We can use the function issecurevariable（tb1， key） to check whether a value that is stored in a table is tainted. The first argument is optional and is by default the global environment \_G. We can thus use the following code to check the integrity of CastSpellByName。

It will print nil followed by the empty string. This second return value is the addon that caused the taint; the empty string means that it was tainted by a /script command entered by the user. As you can see, it is not possible to hook a protected function using the traditional way of hooking it. But there is another way to hook functions

The interface API provides the function hooksecurefunc(tb1, key, hook), which can be used to hook a secure function without affecting its taint. Again, the first argument p^h is optional and is by default the global environment. The last argument is the function that will be executed every time after the hooked function was called. The hook receives the same arguments as the original function

It is actually not a “real” hook, as it’s not possible either to modify arguments passed to the original function or to change its return values. So it is not possible to write an addon like the one I described earlier. But such post-hooks are often sufficient, especially when you want to make minor changes to default UI elements. An example of an addon using such a hook is the Battleground mod of Deadly Boss Mods. It hooks into the function that updates the score frame, and the hook sets the color of the players in the frame to each one’s respective class color

Reload your UI if you haven’t already done this to get rid of our old hook and the taint so we can play around with this function. Let’s test if CastSpellByName is now secure

print(issecurevariable(“CastSpellByName”))

It now prints 1 followed by nil (the mod that tainted the variable if it was tainted), meaning everything is fine. We now apply our new hook, which also just prints the arguments it receives:

hooksecurefunc(“CastSpellByName”, function(...))

print(...)

end)

We can now again check whether it is still secure

We still see 1 and nil as the result, so it’s not tainted. You can now test your spell-cast macros or a simple slash command like /cast Attack; it works properly and the hook function is called

Note that this way of hooking functions works only if the function is stored in a table or global variable (which is technically an entry in a table). You obviously won’t encounter a secure function that is stored in a local variable, as this local variable would only be visible in the file it was created in; this file is digitally signed, and you therefore cannot write your code in it. But you may run into a situation where you need to hook an event handler of a frame without tainting it. You can use the method frame:SetHook(handler, hook) on this frame to add a secure hook to one of its event handlers. We saw this method earlier when we discussed event handlers

You should always use hooksecurefunc or frame:SetHook if you want to hook functions defined by Blizzard that are not tainted yet. The reason for this is that the taint can quickly spread over the whole default UI. A single tainted value can quickly infect other functions and values. Let’s see how taint spreads and how we can trace it

A secure execution loses its secure status and becomes tainted when it executes a tainted function or accesses a tainted value. A value is tainted when it is created or modified in a tainted execution path. Note that just accessing a secure value from a tainted execution path does not taint it. You always create a copy of a value when you access it (note that the value of a variable that holds a complex object like a table is always a reference to that table, so you get a tainted reference to a secure table and not a tainted copy of the table), and it is this copy that will be tainted. The original value stays secure if it was secure before

A single tainted variable can thus quickly spread over the whole default UI as it taints the execution path from the time it’s accessed. All values that are created by this code after the taint occurred will then also be tainted. Other code then accesses these values and also becomes tainted, and so on. You can quickly break something you did not expect to break by tainting a single value

So far we have used a lot of code that was directly called by Lua code of the default UI, like slash command handlers or dropdown menus. But they didn’t break anything, because Blizzard took care of the taint issue in its code. This code either makes sure that no important values can become tainted or uses the function securecall(func, ...). This function takes another function and executes it with the given arguments, and the old taint status of the execution path is restored after the function returns. That means using this function in your own code is pointless, as the taint status would revert to tainted, and that doesn’t help you

Spreading taint was a big issue in earlier versions of the game, especially in the months after the taint system was introduced. However, Blizzard now takes many precautions to prevent addons from breaking the default UI by hooking random functions that do not handle functionality related to protected functions. All critical parts of the default UI make sure that they do not access variables that might be tainted by addons. Hooking functions that do not handle anything related to protected functions directly without using hooksecurefunc is pretty safe

But you should still use hooksecurefunc when you want to post-hook a secure function. It can still happen that tainting an innocent-looking variable or function breaks parts of the default UI. But Blizzard notices and fixes such issues quite fast. I had an example where tainting the function WorldMapFrame\_Update of the world frame could break the Set Focus option, which is certainly an unexpected result. However, this issue was fixed with patch 3.1.1, and it is now safe to taint WorldMapFrame\_Update

However, if you ever notice that your code has broken the default UI, you should run the following command and reload your UI to get more debugging information:

You can then execute the function being blocked because of your addon, and you will get the file taint.log in your Logs folder after logging out. This file contains additional information about the failed protected call and the call that caused the taint

For example, the following hook into the function UnitHealth taints critical parts of the unit frames, and you must never use such a hook in an addon

Now try to engage a mob and kill it. You will see a message in your chat frame saying that a default UI action was blocked due to an addon. The broken functionality is the target’s target frame of the default UI; it won’t be able to update properly if your target changes its target. This was logged in the taint.log file

Taint issues can be very hard to reproduce with different addon constellations. This means that any addon you have can break this example, even if it seems to be completely unrelated to the actual issue. Keep this in mind when you receive an error report regarding taint from one of your users. and always ask for a list of installed addons and a taint log when trying to debug a taint issue

The log file first tells you which addon caused the taint; the addon is MACRO\_TAIN in this case, because I ran the hook from TinyPad, which uses RunScript(str) to execute the entered text as Lua code. Code loaded through RunScript is considered to be a macro by World of Warcraft, as the command /script also uses RunScript internally. The next few lines show the call stack of the execution path that tainted the variable.

The next lines in the log tell you about the actual problem that occurred. The function TargetofTarget\_Update() tried to perform a protected action but it was blocked as the execution became tainted while accessing the global variable UnitHealth. The next lines show the call stack of the failed call and the actual call that failed: TargetofTargetFrame:Show(). The show method of certain frames that deal with protected actions, like targeting or casting spells, is what is called a protected method, which cannot be called from tainted code while you are in combat. Trying to perform such a call results in a notification in your chat frame and an entry in the taint log. We will discuss protected methods and frames in the next section. We now know what we can’t do; let’s see what we can do. It is still possible to create an addon that casts spells or targets units under certain circumstances

A secure template is a normal XML template that can be used for your frames. These templates already have certain functionality that can normally only be called from secure code, like targeting a party member or casting a spell. Possible uses for these templates are unit frames and action buttons. But they are also considered protected frames, and you cannot call certain methods while you are in combat. Let’s write a simple example mod that shows unit frames for you and your current target.

The game provides a secure template that can be used for this: SecureUnitButtonTemplate. As noted, every frame you create from this template will be a protected frame, and a protected frame cannot do certain actions while you are in combat. Let’s look at the restrictions on protected frames before reviewing the available templates

The following methods of protected frames cannot be called from tainted code while you are in combat. All of these methods still work as they work for normal frames when you are not in combat

Children of a protected frame are not necessarily also protected, so you can add normal frames as children to your secure frame. But all parents and frames you anchor your secure frame to automatically become protected, because if they were not, you could simply show /hide or move the parent or anchor to bypass these restrictions. This can sometimes lead to unexpected problems, as frames might suddenly become protected. So be careful when choosing the parent and anchors of a protected frame. You can use the method frame:IsProtected() to check whether a frame is currently protected

The purpose of these restrictions is to prevent addons like CT-HealMonitor, which existed in 2006 before secure templates were introduced in World of Warcraft 2.0. This addon showed a few unit frames, which were sorted by their missing health; that is, the player with the lowest health was ranked highest. Healers could simply always click on the top player in that list and heal him all the time. You can, of course, still create a mod that displays players sorted by their health, but you can no longer target these players by clicking on them while you are in combat. There was also an addon that automatically chose the best heal spell to cast and the best target for it. Such addons are no longer possible; you can only execute a predefined protected action with an explicit click on a protected frame that was created from a secure template

Protected frames do not have special methods assigned to them, but the secure templates are not tainted and can therefore call protected functions. The different available secure templates provide different predefined functionality that is related to protected functions. Let’s see all available templates to get an overview over the whole system before we start with the example mod

Protected frames do not have special methods assigned to them, but the secure templates are not tainted and can therefore call protected functions. The different available secure templates provide different predefined functionality that is related to protected functions. Let’s see all available templates to get an overview over the whole system before we start with the example mod

There are basically two types of addons that need to make extensive use of secure templates: unit frame mods and action bar mods. Action buttons just require a simple template that allows you to execute actions that would otherwise be blocked when you click them while unit frames require a lot of different templates. The following list shows all available secure templates:

SecureActionButtnTemplate:Can be used to implement buttons that execute restricted functions when they are clicked.

SecureUnitButtonTemplate: A template for unit frames, it is tied to a specific unit that will be targeted when you click on the frame. It can also be shown and hidden automatically if the unit comes into existence or vanishes.

SecureGroupHeaderTemplate: An abstract header for a list of unit buttons, as for displaying a raid group. This frame should not be used directly; it is a template for the following templates:

SecurePartyHeaderTemplate: A templare that can contain SecureUnitButtonTemplate to display your whole group; this template inherits from SecureGroupHeaderTemplate.

SecureRaidGroupHeaderTemplate: A template that can be used for subgroups of a raid; it works similarly to the party header.

SecureGroupPetheaderTemplate: An abstract header for a list of pets, it should not be used directly; if acts as a template for the following templates.

All of these secure templates just provide functionality and no design. You are completely free to style them according to your needs. You can add normal frames as children to these secure templates, and there are no restrictions on these children

Let’s now create our simple example mod that just displays two unit frames: one for yourself and one for your target. We need to inherit these two frames from SecureUnitButtonTemplate, but how do we tie that to a specific unit and how can we tell the frame anything about the restricted actions it should do? Just using a simple variable or function call would taint important code of the template and break it. There is another way to communicate with this frame—through attributes of the secure frame

Attributes of secure frames are set by calling the method SetAttribute(attribute, value), where attribute is a string that identifies the attribute and value the new value. GetAttribute(attribute) can be used to retrieve the current value of the attribute. SetAttribute is one of the protected methods of protected frames, which means we cannot change the attributes while we are in combat. The most important attribute we are going to need for our example mod is the attribute unit of the unit button template. This sets the unit the frame is tied to

We now know enough about templates to make use of them, so let’s start building our example mod here. Create a new folder for our addon and call it MyUnitFrames or something similar; then add an appropriate TOC file and add an XML to that TOC file. We now build a template that inherits from SecureUnitButtonTemplate for our two unit frames

It’s no problem to create a template from a secure template; the created template will then, of course, also be protected. This protection is actually also an XML attribute and it is set in the template SecureFrameTemplate in the file FrameXML\SecureTemplates.xml, which is used for all secure frames. And XML attributes are always inherited from the template. The XML looks like this

You cannot change this to talse in your template, because there is no way for you to remove the protection attribute of a frame or template after it has been protected. Add the standard root element to your XML before we start creating our frame

Add the following to your XML file (between the <Ui> tags) to create a simple template that resembles a simple unit frame. It consists only of a portrait, a font string for the name, and a health bar

We defined only two script handlers in the template; these handlers deal with dragging the frame. Note that StarMoving and StopMovingOrSizing are not protected functions; this means that moving the frame with your mouse while you are in combat is no problem. But moving it via calls to SetPoint does not work in combat

We will also need an OnLoad handler that initializes the frame by calling a few functions; for example, we need to call the method RegisterForDrags there. We also need an event handler for the event UNIT\_HEALTH that is fired every time the health of a unit changes; we will update the health bar there

But defining these two handlers in the template is actually not smart, as the two frames we are going to create from this template are too different. But defining these handlers later in the frames would generate some duplicate code, and as I’ve said, duplicate code is bad. It would certainly be possible to define parts of the handlers in the OnLoad handler and use a Lua file with a few initialization functions to avoid a few lines of duplicate code. But the purpose of this example is to demonstrate a simple unit frame in XML with a few lines of Lua code. These few lines of duplicate code are not worth the additional complexity that avoiding them would add

We’ve now discussed the essentials of secure unit button templates, and we are ready to create the actual two frames from our template. Note that the frames are very simple; they just show a portrait, the unit’s name and level, and a simple health bar. Both frames start by default in the middle of the screen but they can be dragged at any time. You can add your own art as textures and an additional mana bar to get a fully featured unit frame mod. Adding this to the example would have added unnecessary complexity, as we would need to take care of a lot of tasks like coloring it correctly according to the mana type (mana/energy/rage and so on)

Let’s start with the player frame. It needs to perform the following tasks in the OnLoad handler: register click and drag events, set the $parentName to the player’s name, and set $partentPortrait to a texture of the player’s portrait. It also needs to call an initialization function that is provided by the secure template:SecureUnitButton\_OnLoad(self, unit, menuFunc). Here, unit is the unit ID the frame is for; this method will then call self:SetAttribute(“unit”, unit) so we don’t have to do this. menuFunc is a function that will be called when the player right-clicks the player frame. We will use the default player dropdown menu here

The event handler uses the API function SetPortraitTexture(texture, unit), which takes a texture as its first argument and a unit ID as its second argument. This function creates a portrait of the unit and sets it as the texture; it also updates this portrait if the appearance of the unit changes (for example, when you equip a new helmet)

We also need the event UNIT\_HEALTH and an event handler that updates the status bar if the event is called with arg1==“player”. Here’s the code

We now have a fully functional player frame that works just fine. Let’s add a target frame; this is slightly more complicated because we need to deal with changing the target. We can use the event PLAYER\_TARGET\_CHANGED here and update the target’s portrait, name, and health in the event handler. But what happens if we select no target at all? We can’t just call the method self:Hide() on our frame, because hiding a protected frame is not allowed in combat

The game provides what is called a secure state driver to do this. We can call the function RegisterUnitWatch(frame) for our frame to tell the secure template API to take care of hiding the frame. It will then hide the frame when the unit that is set in the attribute unit no longer exists and show it when it comes into existence

Add the following button to your XML to create a target frame. Both the OnLoad and OnEvent handlers are very similar to those of the player frame, so this code should be quite easy to understand. Note that there is no need to set a target’s portrait and name with OnLoad, as you never have a target selected when logging in

Our addon breaks the Set Focus option of the default target dropdown menu. The reason for this is that the default UI function that handles this dropdown menu relies on being secure code and we taint it by creating a tainted wrapper function that calls it. Blizzard did not implement Set Focus as a secure button. In a real addon, you would have to create your own dropdown menu that uses a secure action button template; you will see later in this chapter how secure templates can change the focus target

Both frames are now fully functional (except for the mentioned focus issue). But what happens internally when we click on our frame? We never implemented an OnClick handler here, but the frames still behave like normal unit frames. Let’s take a look at the single predefined handler our frame has: the OnClick handler, which calls the function SecureUnitButton\_OnClick

The function SecureUnitButton is, like all other predefined script handlers and helper functions, defined in the file FrameXML\SecureTemplates.lua

Don’t forget that this function is part of the default UI and it is therefore considered secure code. So it is allowed to call all the protected functions we may not call

The helper function SecureButton\_GetModifiedAttribute is used to get the action you want to do based on the mouse button you pressed and modifier keys on the keyboard like Ctrl. This action is defined in the attribute mod-typeX where X is the button (1 = left button, 2 = right button, 3 = middle button), and mod can be a modifier key on the keyboard like alt, shift, or ctrl. It is possible to use \* instead of mod- to ignore modifier keys. Possible values for this attribute are “target” and “menu”. The default values are \*type1\_=\_”target” and \*type2\_=\_ “menu”, meaning that a left-click targets the unit and a right-click opens the menu. The asterisk at the beginning means that modifier keys are ignored. These default values are suitable for our purpose, so let’s keep them

The function then checks whether we want to open the menu and, if so, whether we currently have an active spell cast that is waiting for a target selection. If that is the case, the spell cast is canceled and the function returns at this point. Otherwise it forwards the call to the OnClick handler of the SecureActionButtonTemplate, as a unit frame shares the basic functionality of an action button; that is, something happens when you click on it. This Onclick handler is quite long and boring, as it covers all possible button actions, like casting spells based on certain conditions, and so on. We will return to action buttons later

You may be wondering if we can’t bypass the restriction that you always need a user input to execute a protected function simply by calling the OnClick handler by hand. You could execute the following function call to fake a mouse click

There would normally be no way for an addon to tell if this was a real click or faked. But recall that the whole handler is defined in the template by Blizzard, and it is secure code. An execution path always starts secure and it stays secure if it goes right into this predefined handler. But your call that fakes the click starts from tainted code, as your code is always tainted, and so the whole execution path is tainted. This causes the action to fail

The next step in creating a complete unit frame addon would be implementing party and raid frames; for those we need the templates that are inherited from SecureGropHeaderTemplate

Group header templates control a group of unit frames like your party or a raid subgroup. They manage everything automatically, like removing players who leave the group or sorting the displayed frames by their name or other criteria. I will not show you a complete example of using these templates here, as it is actually quite hard to test them because you need a raid group that is willing to help you with the testing. It is also not very likely that you are ever going to need these templates, as they are only useful for party and raid frame mods. There are already a lot of different raid frame mods out there, and they are highly customizable. The main actions are restricted by the template anyway, so it’s quite unlikely that there is the need to write a new one as all possible features are already implemented by the ones that currently exist

But let’s still see how they work in theory. You simply create a new template from this template and add a simple header like a text to it that shows the raid subgroup or class the header belongs to. You can then use Table 11-1 to set the attributes of this secure template to define how it is displayed

A comma-separated list of players in your group that will be shown under this header. (This causes the template to ignore the attribute groupFilter).

The template does all the hard work; you just define an XML template for a unit button and set it as the template attribute of the header. The header manages the unit buttons; it creates them from your template and does all the positioning. You also don’t need to worry about dealing with events like players joining and leaving the group; this is all taken care of by the header template. The template does this for you because you would not be able to. You can’t show, hide, or move your unit frames while you are in combat, but players can join and leave your raid while you are in combat. The whole template is secure code and can perform all these actions you are not allowed to do in combat

Table 11-1 just lists the attributes for raid or party frames; pet frames have two additional attributes: useOwnerUnit and filterOnPet. Both are boolean attributes. useOwnerUnit displays a unit frame for the owner of the pet instead of a unit frame for the pet itself, and filterOnPet can be set to use the pet’s name for filtering and sorting instead of the owner’s name

The next important type of secure frame is the action button, which is basically just a button that executes a protected function when you click it

Action buttons make extensive use of attributes to define the actions that are taken when the button is clicked. The attribute type controls which action will be executed when the button is clicked. Table 11-2 shows all possible values for pula. All of these actions refer to protected functions, which could normally not be called from tainted code, but the template is secure code

These are a lot of possible options for our button, but they are all really simple. Most of them just call a protected function when the button is clicked. Let’s see an example of a secure action button

A simple example of a secure action button is the Warsong Gulch battleground mod in Deadly Boss Mods. It adds to the score display a small frame that shows the name of the current flag carrier. This frame is a secure action button, and it targets the flag carrier when you click on it. Let’s see how this is done in this DBM module. You can find this mod in the file DBM-Battlegrounds\Warsong.lua of the DBM

The first thing the mod obviously must do is create a frame from the secure template. This is done in the method CreateFlagCarrierButton, which is called when you join Warsong Gulch

The code creates two frames of the type Buttono that inherit from SecureActionButtonTemplate. It then sets their type attributes to macro. You might have expected the type target here, but target can only target a specific unit ID, and there are no fixed unit IDs for your enemies in a battleground. By contrast, macro can execute an arbitrary macro text like /targetexact\_somePlayer, which means macro provides a greater flexibility

The frames are then anchored to AlwaysUpFrame1 and AlwaysUpFrame2, which are the small score display frames at the top of the default UI. The actual macro text is set by the method ?da\_gBh]c?]nnean. This method is called when the flag carrier changes while the player is not in combat and every time the player leaves combat in Warsong Gulch. The local variable FCarrier holds a table with the flag carrier of the Alliance (1) and Horde (2)

This method ensures that the flag carrier button targets the correct player if that is possible. The text of the frame is changed outside this function, because this strategy works fine while you are in combat, so the targeted unit might differ from the displayed text if the flag is picked up by someone else while you are in combat. But there is no way for us to update the target in combat, because of the restrictions on protected frames

I’m not printing the function that determines the current flag carrier and sets the string. The function is quite long, with about 70 lines, because it needs to parse system chat messages to determine the carrier with string\*match as there is no good usable event. The code that sets the text of the frame is also long and complicated, as it needs to determine the class of an enemy player that picks up the flag. You can’t just call UnitClass on enemies that are out of your range, so the function uses a workaround and accesses the battleground scoreboard, which contains the classes of all enemies. You can read the function that is stored in the local variable updateflagcarrier at the end of the file if you want to know how this works

But an action button can do more than just targeting someone when you click on it

Let’s create a simple example action button to see what this feature is capable of. Create a new addon with an appropriate TOC file and add a new XML file to it. You can also recycle the example unit frame mod we created earlier in this chapter if you don’t want to restart the game for the new addon

We will create a simple action button for testing here; this button will just show up in the center of your screen and use the default style from OptionsButtonTemplate. We can then use an in-game editor like TinyPad to execute short Lua scripts to set attributes to test our button

Creating a secure button with the style of a default action button is really easy thanks to multiple inheritance. We can just add both SecureActionButtonTemplate and OptionsButtonTemplate to the inherits attribute. Or more generally, you can always just add a secure template to your inherits attribute if you need the functionality of a secure template. The XML file then looks like this

This code displays a small button in the center of your screen with the text “Secure Button.” Feel free to place the button wherever you want to. The call to RegisterForClicks in the OnLoad handler is important because by default the button listens only to clicks with the left mouse button, and we want to use the right button later in this example

We will now use advanced attributes to do something with this button. We can access it under the global variable SecureTestButton, so let’s add the first few more advanced attributes to it

In the unit frame example, you’ve seen that attributes can contain modifiers that allow you to execute different actions based on the mouse button that was pressed. A modifier of an attribute consists of a prefix and a suffix, where the prefix holds modifier keys on the keyboard while the suffix defines the mouse button. Both the prefix and suffix are optional; the default value matches all keys

A good button type for testing is always macro, as you can just assign something like /say hello, world as the macrotext attribute. Execute the following code to set the button’s type to macro

Let’s now add macro texts depending on the mouse button used. Earlier we saw the prefix \* in the default values from the unit frame template; this is actually not always necessary. Omitting the prefix and suffix has the same effect as using an asterisk for both. However, omitting just the prefix or the suffix means that none of the modifier keys or mouse buttons may be pressed. It is obviously not possible to click an onscreen button without using a mouse button (using another button with the type “click” to fake a click sends the mouse button the screen button was clicked with), so you don’t want to omit the suffix when you use a prefix

The following code sets the macrotext attribute that is used if the button is clicked with the left mouse button (recall that we can append 1 to the attribute name to tie it to the left mouse button); we don’t need the asterisk for the prefix here

Your character now says “hello” when you left-click the button. A real example mod would say “hello, world,” so let’s add “world” as right-click text

We can now also add a keyboard modifier. We need to use the asterisk as a suffix here, or the action would match no mouse button

Our character now says “hello world” when we shift-click the button with any mouse button. Note that the old functionality didn’t break; we can still use just the left or right button to get “hello” or “world” respectively

It is also possible to remove a specific modifier combination by setting its value to the value of the variable ATTRIBUTE\_NOOP (noop is short for “no operation”; the variable actually just holds the empty string). Using Shift and the middle mouse button currently results in “hello world”; let’s remove that.

It is possible to use these modifiers for all attributes including the type attribute. A button can then execute completely different actions based on the used mouse button and modifier keys. Let’s try this by changing the type to p]ncap when you right-click the button with the Alt key down

A good target unit for testing is player, so let’s set this as the unit attribute

You can now target yourself by alt-right-clicking the button. Note that the secure templates will always evaluate the type attribute with all its modifiers before it gets the required additional templates. This means that adding the attribute alt-macrotext2. is pointless now as it would never be used because the type will always be target when you right-click the button with the Alt key down

The whole example seems to be highly artificial, as you certainly never need a button that sends text based on the mouse button used. And sending chat messages can also be implemented without using secure frames at all. We will see more useful and powerful command related to secure actions in the next chapter when we discuss macros. Macros can be used to implement functionality very similar to secure buttons; they can even execute the OnClick handlers of existing secure buttons

This chapter first showed how the game prevents addons from automating the gameplay with the code taint system. This taint system makes sure that unmodified code of the default UI has more permissions than code that is provided by the user. We learned about protected functions, which cannot be called by our code. But we also saw how addons may circumvent these restrictions in certain situations by using secure templates

The last part of the chapter was about implementing advanced secure button actions by using modifiers in attributes. You saw how we can create a single button that executes completely different actions based on the used mouse button and keyboard modifiers. Note that everything behind the secure templates, including the modifiers in attributes, is implemented in Lua. You can read the file FrameXML\SecureTemplates.lua of the default UI if you are curious how these templates work internally. This file is one of the more complicated in the default UI, but it is also one of the best commented, so it should be understandable.