Iriel's Field Guide to Secure Handlers

An introduction and reference to the new WoW 3.0 Secure Handlers and their use.

Iriel (Silver Hand) - August 2008

Introduction

The Burning Crusade expansion (aka WoW 2.0) brought about some significant changes to the add-on environment, the most impact almost certainly coming from the restrictions on actions and on modification of frames in combat. In order to still permit add-ons to provide interactions with protected frames the SecureStateHeader was conceived, offering some control over protected frames.

While the SecureStateHeader certainly served its purpose in enabling some functionality while being tightly restricted to prevent abuse, those who have used it (and perhaps even more so, those who tried but failed to use it) are aware that it is at best cryptic, and also severely limited. The Wrath of the Lich King (WoW 3.0) completely replaces the SecureStateHeader that aims to be both far more intuitive as well as more powerful.

This guide aims to describe the features of the new system, as well as a reference for the capabilities it offers.

The Design Goals

Before embarking on the description of the system, it's worth laying out some of the design goals that shaped the implementation:

The secure headers intend to:

- allow add-on code to perform certain otherwise-protected actions on frames during combat.
- define the actions in the most intuitive and familiar manner possible.
- allow any 'decision making' that would be available via the macro conditional system.
- allow all functionality previously available with the SecureStateHeader.
- introduce as little overhead as necessary.

Similarly, they intend to prevent:

- interference/control from insecure code during combat.
- access to information about the state of the world beyond that available through macro conditionals.

- unsafe manipulations of frames during combat.
- invocation of functions beyond those explicitly permitted for use.

A Quick Overview

In order to manipulate protected frames in combat, you begin by creating a Frame that inherits from one of a number of Secure Handler templates. This frame is known as a 'header' or 'owner' frame.

All secure handlers have some built in behaviors in response to setting of certain frame attributes, in addition each of the handler templates provides support for different triggers (such as a click, or mouse enter/leave, or show/hide, etc).

For each of these triggers you specify an action or actions to perform by writing snippets of normal lua code, this code has access to most of the normal lua functions and some of the WoW functions, as well as the ability to control protected frames (such as itself and its children). In addition each 'owner' frame has its own variable scope that all of its snippets are executed in, so you can pass data between them, and maintain state, simply by reading and writing global variables.

The code for each handler is specified by setting a specially named attribute on the owner frame (or on its children), whose value is a String containing lua source code (You cannot pass closures directly as they may be referencing unapproved functions or data).

Key Concepts

There are some key concepts behind what was just described, each of which will be explained in more detail later. These are:

- Secure handler instances are Frames that inherit from one of the **Secure Handler Templates**.
- Each secure handler instance, or owner frame has a **Restricted Namespace** containing those parts of the lua and wow API's that are permitted, and a global environment to store and share data which is a **Restricted Table**.
- Secure Header actions are described by writing lua code, subject to some small limitations, that internally is used to create **Restricted Closures** – secure code compiled from user provided lua source that runs in a tightly managed 'sandbox'.
- The snippets of code can manipulate themselves and other protected frames using **Frame Handles**, which are objects that behave somewhat like normal frames but only allow permitted operations.
- In addition to the direct owner actions, there are also **Secure Handler Child** Templates and features that provide interaction between other frames and the owner.

The Secure Handler Templates

Base Templates

All secure handler templates are derived from one of two base templates, each of which provides a special **OnAttributeChanged** script handler to support the core functionality.

SecureHandlerBaseTemplate

This is the default base template for all secure handlers, it supports the following special attributes:

owner:SetAttribute("_execute", "*snippet*")

When the "_execute" attribute is set to a string value, the value is used as a snippet of lua code, which is compiled and executed as a restricted closure within the owner's environment. This mechanism is intended primarily to populate or change the contents of the environment, but can be used for other purposes.

The _execute attribute is automatically set back to nil as part of the operation.

owner:SetAttribute("_frame-*<label>*", *frame*)

Setting any attribute beginning with "_frame-" and with one or more characters following the dash will cause the value to be tested to see if it's a protected Frame. If so, then the attribute **"_frameref-*<label>*"** will be set to the frame handle for that protected frame. For any other value of *frame* the frameref attribute will be set to nil. This mechanism is provided to allow a secure handler to be given control over any protected frame. See later sections for details on frame handles (TODO – Actually cover it)

The _frame attribute is automatically set back to nil as part of the operation.

owner:SetAttribute("_adopt", *button*)

Setting the "_adopt" attribute with the value of a frame that has an OnClick script handler allows the secure handler owner to be called whenever that frame is clicked upon. This allows for remapping of buttons, as well as potentially taking actions before or after the click. This functionality is covered in more detail below (TODO – Actually cover it)

The _adopt attribute is automatically set back to nil as part of the operation.

SecureHandlerStateTemplate

This template is an extension of the SecureHandlerBaseTemplate that supports all of the attributes above in addition to the following:

owner:SetAttribute("state-*<stateid>*", *value*)

Whenever an attribute is set whose name begins with "state-" followed by one or more characters then a snippet of code can be executed to react to the state notification:

Snippet Attribute:	**_onstate-*<stateid>***
Parameters:	**(self, stateid, newstate)**
	self = the owner frame
	stateid = the id of the state attribute that changed
	newstate = the new value for the attribute
Returns:	*none*

The corresponding **"_onstate-*<stateid>*"** attribute is queried, and if it is a string then the string is used as a snippet of lua code which is executed with the parameters **(self, stateid, newstate),** where self is the owner frame, stateid is the stateid component of the attribute name, and newstate is the value set. This allows state-dependent capabilities like those from the original SecureStateHeader, and is particularly useful in conjunction with the SecureStateDriver mechanism.

Action Templates

These templates each extend the SecureHandlerBaseTemplate but each adds a different set of script handlers, you can combine them as necessary to support interaction at the handler owner level.

TODO describe the child update propagation and standard arguments here so I don't have to keep repeating it.

SecureHandlerClickTemplate

This template simply provides support for the OnClick frame action.

<OnClick>

Snippet Attribute:	**_onclick**
Parameters:	**(self, button, down)**
	self = the owner frame **button** = the button which was pressed **down** = flag indicating if this was a press or release
Returns:	**childupdate, childmessage, animate**
	childupdate = flag or string if child update snippets are to be called afterwards. **childmessage** = message parameter to pass to child snippet **animate** = set to a true value to enable animation handlers (SUBJECT TO CHANGE)

...

SecureHandlerShowHideTemplate

...

<OnShow>

Snippet Attribute:	**_onshow**
Parameters:	**(self)**
	self = the owner frame
Returns:	**childupdate, childmessage, animate**
	The standard child action return values.

<OnHide>

Snippet Attribute:	**_onhide**
Parameters:	**(self)**
	self = the owner frame
Returns:	**childupdate, childmessage, animate**
	The standard child action return values.

SecureHandlerEnterLeaveTemplate

...

<OnEnter>

Snippet Attribute:	**_onenter**
Parameters:	**(self)**
	self = the owner frame
Returns:	**childupdate, childmessage, animate**
	The standard child action return values.

<OnLeave>

Snippet Attribute:	**_onleave**
Parameters:	**(self)**
	self = the owner frame
Returns:	**childupdate, childmessage, animate**
	The standard child action return values.

The Restricted Namespace

Quick blurb on what this is (set of allowed functions)

High level summary of what's in and what's out

Brief mention of restricted tables and special table methods

Read-only to executing code

Restricted Tables

Why they're needed

The restrictions themselves (secure modify, limited datatypes)

Creating them

Accessing outside of secure code (rtable)

Restricted Closures

Limitations (no closures, no direct table creation)

Secure code in a sandbox

Cached by body and parameters

Frame Handles

Proxies for actual frames

Only allowed for explicitly protected frames

Method summary (with restrictions)

Secure Handler Child Support

Child handler templates

Other tricks

Blah

Examples?

Some simple and not so simple examples??