

**HEAT** Next Generation

Plug-in:

Native Interception

###### 

###### 

|  |  |
| --- | --- |
| **Date:** | 3rd Mar 2008 |
| **Customer:** | Holodeck Next Generation |
| **Author:** | Rahul Chaturvedi (rchaturvedi@securityinnovation.com) |

Security Innovation, Inc.

187 Ballardvale Street, Suite A170

Wilmington, MA 01887

+1.978.694.1008  
  
www.securityinnovation.com

# Background

The Native Interception plug-in will provide the functionality to intercept native calls. This plug-in will also setup an insertion point on LoadLibraryExW along with the setup component and the communications server.

# Basic Design

Intercepts provided to the Native Interception plug-in need to be able to get the virtual addresses of function entry points of the functions we are intercepting and the virtual addresses of the intercept functions for them. The user can either specify a virtual address directly, or give . The Intercept functions are responsible for maintaining the stack!

These are the various components and what they'll be responsible for;

1. Setup entry-point: The setup entry point will take a list of interception function objects. These interception function objects need to provide an original function identifier object and an interception function identifier object. These objects will contain either virtual addresses, or a module name and a function name. The setup function will then go ahead and setup all functions except functions whose original function object cannot be resolved (in case the original function module isn't loaded)
2. LoadLibraryExW insertion point: This insertion point will monitor for any loads of modules which exist in the deferred original function modules list. If one shows up, it will intercept the requisite function and remove the deferred mark from the intercept
3. Communications interface: The communications interface will provide the ability to enable/disable intercepts and give back diagnostic data
4. Native Interception Layer: This layer is shared with the base HEAT NG framework. It provides basic interception for native functions based on virtual addresses

# Design Details

## Setup

This will essentially just be a call from the HEAT NG framework. These are the tasks that the setup entry point will perform;

* Create an internal list of interception objects
* Create the comms server
* Resolve all the objects in this list and call the native interception layer to perform interception on them
  + Unresolved objects will be marked as deffered
  + Objects that throw errors will be removed from this list and an error will sent back to HEAT NG
* Start the comms server and return

## Insertion point

This is the function of this insertion point at LoadLibraryExW

* Check name of module against deferred functions list
* If the module exists, pull out the deferred functions matching the module being loaded
* Call the native layer to perform the interception
* Call the original LoadLibraryExW function (provided as a parameter)

## Communications Interface

The comms interface will provide the functionality to accept enable/disable commands for intercepts and provide diagnostic feedback

* To enable an intercept, the comms interface will simply 'intercept' a function
* To disable an intercept, it will 'unintercept' the function

Since we're working with instructions directly, we can use the interlocked instructions to allow this to happen in a thread safe way

## Native Interception Layer

This layer will be shared with the base HEAT NG Framework. This layer will provide two calls; intercept and unintercept. Both calls will take two virtual addresses; one for the original function, one for the interception function. The design of this layer needs to be bordering on ridiculously simple. This code needs to be as efficient as possible, which eliminates any possible bookkeeping (as we have in the current HEAT) and any complex interception function code. This is how the two functions are planned to be implemented;

* Intercept
  + This call will first create a replacement header for the original function call
  + Then it will create an imposter function. This is what the imposter function would look like;
    - Remove Return Address from the Stack
    - Push Replacement Header Address onto the stack
    - Push Return Address back onto the stack
    - Perform a "call" to the interception function
    - [All the address will be hard coded into the instructions themselves]
  + Then it will overwrite the first five bytes of the original function call with a jmp instruction to the imposter function
* Unintercept
  + This function will remove an imposter function by,
    - Getting the address of the replacement header by reading the address in the second instruction of the imposter function
    - Replacing the replacement header back onto the original function

To remove the need to store additional data simply for keeping imposter function and replacement header address information, we'll end the replacement header with a canary value. This will be a sequence of bytes which "cannot" ever execute and will signify the end of the replacement header.

This mechanism is going to be generations simpler than what is currently implemented in HEAT.

Eventually the Native Interception Layer will grow with more API's to provide miscellaneous native functionality. This will be on a need basis.