HEAT: Runtime interception of Win32 functions

Michael M Andrews
Computer Science Department
Florida Institute of Technology
150 West University Blvd.
Melbourne, Florida 32904
mike@se.fit.edu

Introduction

When researching in any field, it is always important to build upon existing work. In most research disciplines, the core base is well defined and widely documented. However, the basis of innovative research in computer science, especially at the systems level, can often be propriety information. This can be a major problem as unless you have source code available you must be granted access to the propriety information (by usually signing non-disclosure documents) which comes with barriers on access, ownership and the ability to publish results.

As an example, Microsoft's Windows operating system (all versions) can be a good base for research as it is a "real world" environment with rich usage patterns and any potential advances made would benefit a large number of people. However, access to the Windows operating system is closely guarded via a number of API's (Application Programming Interfaces) along with the fact that Microsoft jealously guards it source code[1]. Therefore, the ability to easily tie into and extend the functionality of applications and operating systems where source code is not available is a necessity.

With this in mind, researchers at the Florida Institute of Technology created HEAT – Hostile Environment for Application Testing. The goal of this research was initially to create a tool that would create hard to reproduce environments (like low or corrupt memory, insufficient disk space, slow network, etc) to help uncover defects in software. To achieve this it was clear that HEAT would have to tie into both the operating system and the application under test.

Interception technique

The general term used for tying into existing functionality is *hooking* for which there are a number of techniques available. HEAT uses a technique known as binary code rewriting which, as shown in figure 1, works by rewriting a function's instructions to call an "imposter" function that executes instead of the original function. Other interception techniques work by simply overwriting the original function with imposter code but it is important to preserve the original functionality so that it can be utilized later in the execution if required.

[1] Access to the Windows NT source code is available to select researchers after signing Microsoft Windows NT source code access license.

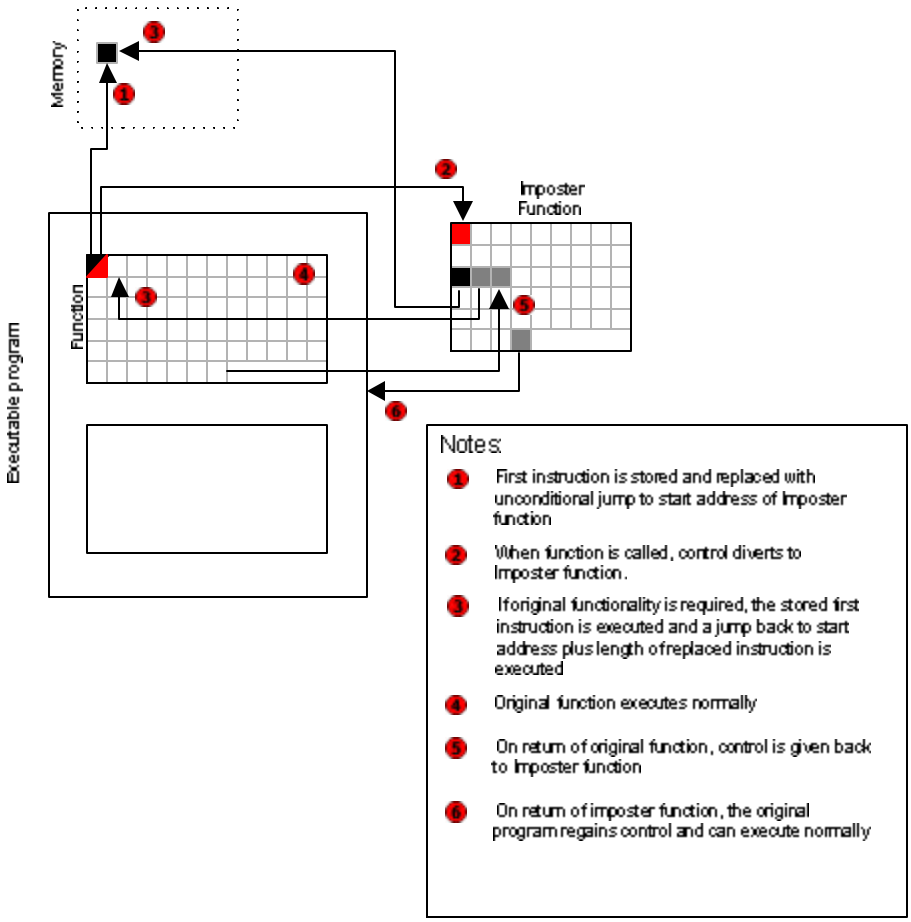


Figure 1 Binary code rewriting

Architecture

The architecture that HEAT is based on is a single dynamically linked library file (heat.dll) and an API. To use HEAT, a program would include the API (which is a C++ class called CHeatApp), have the API's precompiled source code as a dependency (HeatApi.lib), and have the main library (Heat.dll) available. It is then a simple task of creating a CHeatApp object and calling the relevant member functions to create or attach to the target process that we want to intercept functionality. We will discuss HEAT's API functions later in this document but the next sections are dedicated to how the API and the library communicate as well as the library's main tasks.

When the API is initialized, one of the main tasks is to load the HEAT library. Within the library is the mechanism to intercept functions and a lookup table of all the functions so far intercepted (see later). To avoid any mechanisms the operating system might have in protecting code from modification, the HEAT library must become part of the process we

wish to intercept so that it has the same privilege level. This technique is known as dll injection and is documented by Jeffrey Ritcher in his article "Load Your 32-bit DLL into Another Process's Address Space Using INJLIB" (Microsoft Systems Journal, May 1994). Once loaded, the library then has to form a communication path back to the HEAT API. This is achieved by creating a named pipe called "HeatDataPipepid" where pid is the process ID of our program and allows for multiple copies of HEAT to be active without interfering with each other. Across this communication pipe, the HEAT API sends commands to the library about which functions to intercept or to withdraw (unintercept) from.

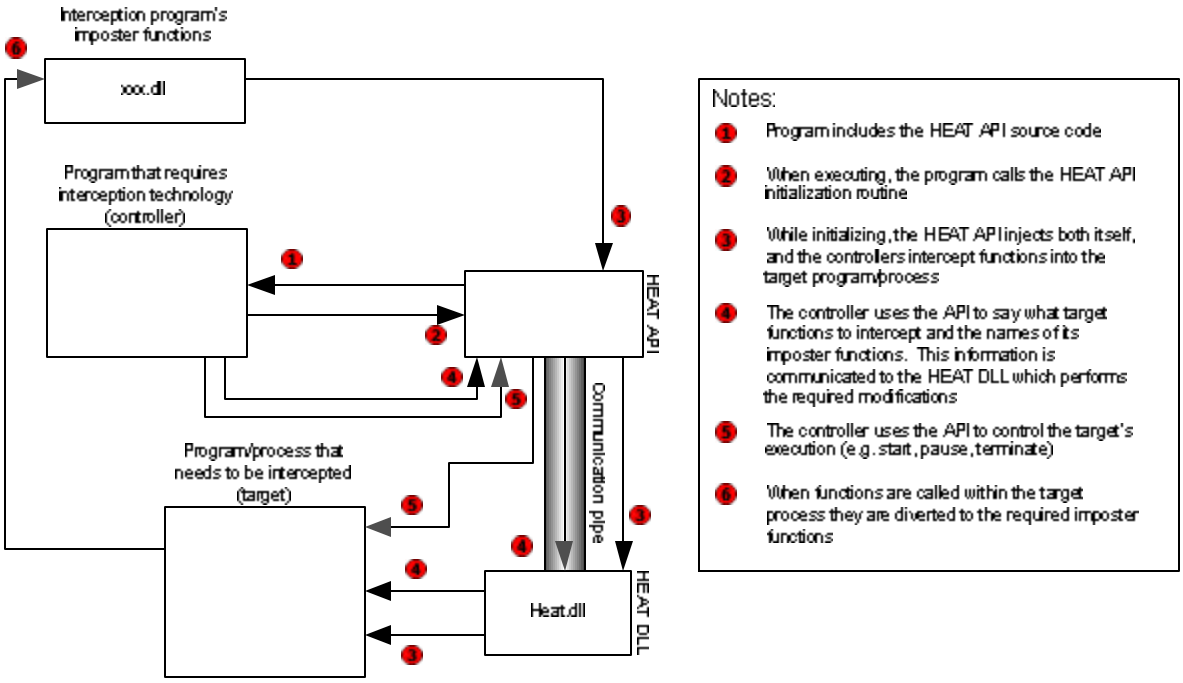
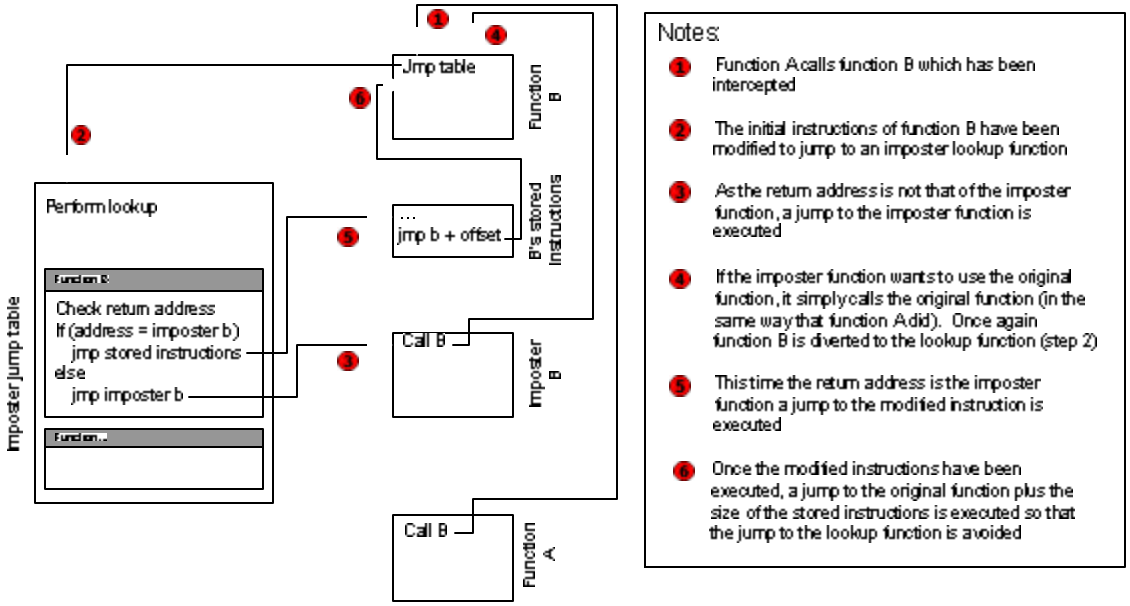


Figure 2 HEAT architecture

In order to intercept a function, we perform binary modifications on the instructions in a similar method shown in figure 1. However, this method assumes that all instructions are the same size and that we know the addresses of both the original function and the stored instruction. Firstly, if we consider a mechanism whereby we store the addresses of all the functions intercepted along with their saved instructions, we have a solution whereby a single function that utilizes a jump table can be used to divert execution off to the required imposter function. This enables us to access the original function from our imposter function by simply calling its function again and not having to remember any addresses or alias function names. Further, as all the interceptions are achieved via jump instructions, the stack is preserved and function parameters are available by name as they were in the original function.

Finally, as instructions on the Intel processor are not a fixed size, simply replacing an arbitrary instruction with a jump instruction is not a trivial task. To achieve this we look at the first 5 bytes[2] of the original function to see if it ends on an instruction boundary. If we are not on an instruction boundary, we simply carry on reading a byte at a time until a boundary is discovered. At this point we can store the original 5 bytes (along with any additional bytes up to the instruction boundary) and rewrite it with a jump instruction into our interception jump table.



The next step required is to load the HEAT library into the target process so that it can perform the binary modifications required to intercept arbitrary functions. Depending on the processes status, the functions `delayedInject` (when the process in uninitialized, e.g after `initializeApp` is used) or `forceInject` (after the process has been created e.g. when `attachToApp` has been used) will perform the injection.

Often, the next step is to perform the interceptions. This is simply achieved with the function `interceptFunc` which takes four parameters: the function to intercept, the library that the function is within, the function that we wish to call instead (our imposter function) and finally, the library that our imposter function is within. It is possible to omit the 2nd parameter, whereupon HEAT will search through all the libraries loaded and intercept all functions with the given name.

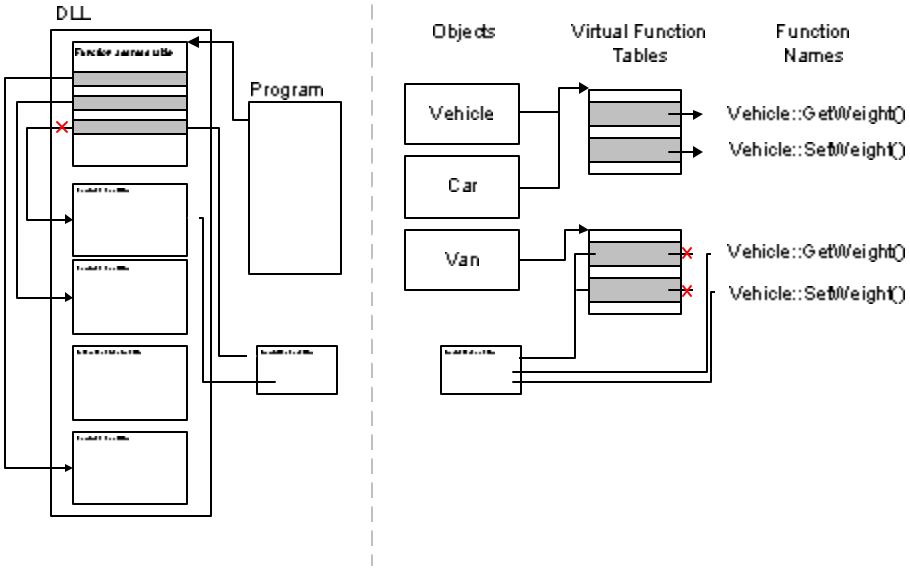
After all the interceptions have been specified, it is time to run the program. HEAT provides four functions for controlling the program – `runApp`, `pauseApp`, `resumeApp` and `terminateApp` – which are all quite self explanatory.

As all the interceptions on a program are carried out in memory, when the program has finished or we want to stop it, it is quite acceptable to let it terminate naturally or via `deinitializeApp`. This is because any interceptions are performed on an execution-by-execution basis and therefore modifications are temporary for a given execution. However, sometimes it is desirable to withdraw any interceptions gracefully from a program, for example when the target program has to carry on running (i.e. a process that should not be shutdown like explorer or a service like IIS). Interceptions can be removed singularly using `unInterceptFunc`, passing once again the first two parameters from `interceptFunc` to un-intercept the function. The final task is to remove the HEAT library from the process by calling the function `detachFromApp` which will remove any remaining interceptions before completing.

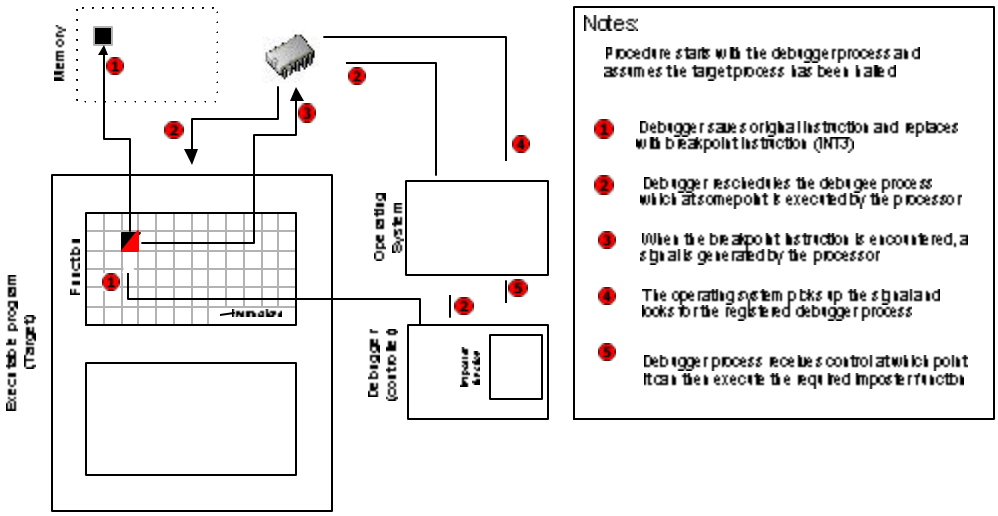
Related work

The idea of tying into existing functionality, or hooking, is not entirely new [refs]. A previous, similar technique known as *code patching* has existed since the beginnings of digital computing when modifying a program's binary source was considered more practical than recompiling the entire application [ref]. Modern tools employing these techniques include *EEL* [ref] and *Etch* [ref] which generally take a binary program and an instrumentation script to produce a new, instrumented binary. However, this technique is often not suitable as modifications can only be undone by restoring the original binary. The only exception within this technique is *DyninstAPI* [ref] which performs binary rewriting dynamically. However, because of the insertions into the binary, the stack and register values are not always preserved, therefore making calls to the uninstrumented function difficult.

Of the available techniques remaining, import address redirection is the most common and receives the most discussion [refs]. The basic technique, as show in figure 4, is to alter the DLL's import address table – the start address lookup table for exported functions within the DLL. Therefore, when a function within a redirected DLL is called, the start address of the imposter function is used instead of the address of the original function. This technique works equally well with object-oriented code by modifying the program's virtual function table – a table that maps functions that can be overridden to their code base location.



fully-fledged debugger, this process could simply execute the imposter function and resume the target program.



Conclusions

As we have seen, the HEAT API is both a powerful *and* an easy to use interception mechanism. Its success has been clear: over the past 4 years, HEAT has been a basis for a number research projects at Florida Tech, has been incorporated into tools used extensively at Microsoft, Rational, IBM, and has evolved into a mature technology.