A C++ Style of Intercepting Functions

By **armagedescu** | 13 Nov 2010

This article is about detouring functions in a more safe C++ programming style.

- Download source - 4.06 KB

Introduction

If you write a patch to a function, it is easy to make mistakes. The compiler can't help you to find these kind of mistakes. You will find most of them usually as a crash at runtime with some ugly error box. And even worse, it can be dangerous for operation system, when you patch a function globally. This is why I've created a class that helped me to do this more safely. I'm not quite sure what rewriting of executable code can be safe, but I think some problems could be avoided at compile time.

At another point, you may want to set a patch when executing some function, to remove the patch when executing some other function, to remove automatically the patch when exiting the function, to set the patch forever, to apply the patch immediately or to prepare it to set it later.

It is a bit funny to talk about safe rewriting of executable code, or even more, to use OOP when intercepting functions. :) But you will be surprised, this works. C++ templates, encapsulation and automatic destruction really makes this much easier.

Using the Code

You can use this code just by adding the *patcher.cpp* and *patcher.h* to your C++ project. The archive does not contain samples.

Some Background

There can be three types of such mistakes (let's call them **3TM**):

1. Using some wrong calling convention, what is not the same as the one of patched function
2. Using wrong arguments passed to function
3. Using wrong return type

It is very easy to avoid these kind of mistakes at compile time.

Let's take a look at functions `send`/`recv`in *winsock.h*:

```
int PASCAL FAR recv (
                  IN SOCKET s,
                  OUT char FAR * buf,
                  IN int len,
                  IN int flags);
int PASCAL FAR send (
                  IN SOCKET s,
                  IN const char FAR * buf,
                  IN int len,
                  IN int flags);
```

And the simplest patches for them:

```
int (PASCAL FAR *lpfn_send )( IN SOCKET s, IN const char FAR * buf,
              IN int len, IN int flags);
int PASCAL FAR my_send ( IN SOCKET s, IN const char FAR * buf, IN int len, IN int flags)
{
```

```
   return lpfn_send (s, buf, len, flags);
}
int (PASCAL FAR *lpfn_recv )( IN SOCKET s, OUT char FAR * buf, IN int len, IN int flags);
int PASCAL FAR my_recv ( IN SOCKET s, OUT char FAR * buf, IN int len, IN int flags)
{
   return lpfn_recv (s, buf, len, flags);
}
```

The rules are simple. If some function `send` has some type `type_of_send`, then the patch function `my_send` must use the same type `type_of_send`. The trampoline function `lpfn_send` must have type `type_of_send` as well. If some other function `recv` has some other type `type_of_recv`, then the patch function `my_recv` must use the same type `type_of_recv`. The trampoline function `lpfn_recv` must have type `type_of_recv` as well.

And the patching itself (this is a theoretical code yet):

```
patch(send, my_send, lpfn_send);
patch(recv, my_recv, lpfn_recv);
```

See no forced type casts, no warning, no errors. Much easier to write, and no chance to make **3TM**.

The prototype of this function is:

```
template<class T> patch(T, T, T&);
```

This construction helps you to avoid the **3TM**. If you don't enforce type cast when passing arguments to function, then you will not be able to compile code with **3TM**.

Not a big difference when using pointers instead of references:

```
patch(send, my_send, &lpfn_send);
patch(recv, my_recv, &lpfn_recv);
```

The function prototype is:

```
template<class T> patch(T, T, T*);
```

This is all about safety of patching.

Function Typing

Something about rules of function types. You may need this, when you get the address of a function via `GetProcAddress`. Let's take a simple C++ function declaration:

```
int func(int, char*, void*);
```

The type `T` of function will be:

```
int (*)(int, char*, void*)
```

You have to declare a pointer of correct type, and cast the type correctly when doing `GetProcAddress`:

```
int (*func_ptr)(int, char*, void*);
func_ptr = (    int (*)(int, char*, void*)    ) GetProcAddress(hModule, "func_name");
```

A more complete function declaration is:

```
return_type calling_convention function_name (list of types of arguments);
```

The rule of creating function type is:

```
return_type (calling_convention *) (list of types of arguments)
```

The default calling convention for C++ in MSVC function is `__cdecl`. You may omit this. But you may include it in the declaration. And you must include calling convention when you are working with functions with different calling conventions. The above sample function with pointer declaration and the `LoadLibrary`with type cast becomes:

```
int __cdecl func(int, char*, void*);
int (__cdecl* func_ptr)(int, char*, void*);
```

```
func_ptr = (  int (__cdecl*)(int, char*, void*)   )
      GetProcAddress(hModule, "func_name");
```

The same thing for functions send/recvsample looks like this:

```
//functions declarations in winsock.h
int PASCAL FAR recv (
                  IN SOCKET s,
                  OUT char FAR * buf,
                  IN int len,
                  IN int flags);
int PASCAL FAR send (
                  IN SOCKET s,
                  IN const char FAR * buf,
                  IN int len,
                  IN int flags);

///////////////////////////////////////////////
//pointer and patch declarations in our application
//let's omit IN and OUT
int (PASCAL FAR *lpfn_send )( SOCKET , const char * , int , int) = send;
int PASCAL FAR my_send ( SOCKET s, const char * buf, int len, int flags);
int (PASCAL FAR *lpfn_recv )( SOCKET, char FAR *, int, int) = recv;
int PASCAL FAR my_recv ( SOCKET s, char FAR * buf, int len, int flags);

///////////////////////////////////////////////
//pointer assigning with GetProcAddress:
HINSTANCE hInstanceWs2 = GetModuleHandleA("ws2_32.dll");
lpfn_send = (int(PASCAL FAR*)(SOCKET,const char*,int,int))
          GetProcAddress(hInstanceWs2, "send");
lpfn_recv = (int(PASCAL FAR*)(SOCKET,char*,int,int))
          GetProcAddress(hInstanceWs2, "recv");

//I don't advise to do this, but is ok, not a big deal:
lpfn_send = (int(__stdcall*)(SOCKET,const char*,int,int))
          GetProcAddress(hInstanceWs2, "send");
lpfn_recv = (int(__stdcall*)(SOCKET,char*,int,int))GetProcAddress(hInstanceWs2, "recv");
```

In most cases for WinAPIfunctions, you don't need to call GetModuleHandle/GetProcAddress, because libraries and functions are loaded to some default addresses. So, you don't need the above type casts. Just assign function address in initialization or later:

```
lpfn_send = send;
lpfn_recv = recv;
```

Real World

It's time to use real code, see *CPatcher.zip*. I have created a simple class named CPatch. Now, the two constructors of the class are quite similar with the theoretical function patch(...) described above:

```
class CPatch
```

```
{
...
CPatch(){}
CPatch(CPatch&){}
...
public:
...
template<class TFunction>explicit CPatch
   (TFunction FuncToHook, TFunction MyHook, TFunction& NewCallAddress,
           bool patch_now = true, bool set_forever = false)
...
template<class TFunction>explicit CPatch
   (TFunction FuncToHook, TFunction MyHook, TFunction* NewCallAddress,
       bool patch_now = true, bool set_forever = false)
...
template<class TFunction>explicit CPatch(TFunction& NewCallAddress,
   TFunction MyHook, bool patch_now = true, bool set_forever = false)
...
template<class TFunction>explicit CPatch(TFunction* NewCallAddress,
   TFunction MyHook, bool patch_now = true, bool set_forever = false)
...
```

How To Use

Let's see some ways we would like to use patches.

4 Apply the patch immediately locally, and remove it automatically when you exit function. The destructor of C++ class works very well:

```
#include<winsock2.h>
#include "patcher.h"
int (PASCAL FAR *lpfn_send )( IN SOCKET s,
   IN const char FAR * buf, IN int len, IN int flags);
int PASCAL FAR my_send ( IN SOCKET s, IN const char FAR * buf,
   IN int len, IN int flags)
{
   return lpfn_send (s, buf, len, flags);
}
int (PASCAL FAR *lpfn_recv )( IN SOCKET s, OUT char FAR * buf,
   IN int len, IN int flags);
int PASCAL FAR my_recv ( IN SOCKET s, OUT char FAR * buf,
   IN int len, IN int flags)
{
   return lpfn_recv (s, buf, len, flags);
}


....
{
```

```
  CPatch patch_for_send(send, my_send, lpfn_send);
  CPatch patch_for_recv(recv, my_recv, lpfn_recv);
  //now the functions send and recv are patched
  .....
}//the patches are removed automatically
```

You may try to make some **3TM** to see what compiler will not allow you to do this. Try to use something else instead of PASCAL. Or try to change the type of an argument. Or try to use `recv`instead of `send`by mistake:

```
patch_for_send(send, my_recv, lpfn_send);
```

Perhaps you've noticed two pairs of constructors. One is using references. But another is using pointers. So, if you like to use pointers, you do like this:

```
CPatch patch_for_send(send, my_send, &lpfn_send);
CPatch patch_for_recv(recv, my_recv, &lpfn_recv);
```

Inside both constructors calls, the same nontemplate and protected function `HookFunction`. Only difference is, in the first variant, is used `&NewCallAddress`, and in the second `NewCallAddress`.

You may initialize the trampoline pointers to real functions, i.e. `lpfn_send = send`, and `lpfn_recv = recv`:

```
int (PASCAL FAR *lpfn_send )( IN SOCKET s, IN const char FAR * buf,
       IN int len, IN int flags)
                  = send;

int (PASCAL FAR *lpfn_recv )( IN SOCKET s, OUT char FAR * buf, IN int len, IN int flags)
                  = recv;
```

In this case, you can use other two constructors, that is even shorter:

```
CPatch patch_for_send(lpfn_send, my_send);
CPatch patch_for_recv(lpfn_recv, my_recv);
```

Now, you may apply and remove patch as many times as you like.

```
  //we do not need patches
  patch_for_send.remove_patch();
  patch_for_recv.remove_patch();
  //we need the patches right now! :)
  patch_for_send.set_patch();
  patch_for_recv.set_patch();
  //we do not need patches
  patch_for_send.remove_patch();
  patch_for_recv.remove_patch();
  //we need patches again
  patch_for_send.set_patch();
  patch_for_recv.set_patch();
}
//if you've forgotten to remove some patches, not a crime.
//Destructor does this automatically for you
```

If you want to remove the patch forever, pass `true`to default parameter, what is implicitly set to `false`:

```
patch_for_send.remove_patch(true); //true = 4ever
```

Let's now use a prepared patch. It is not applied immediately:

```
CPatch patch_for_send(send, my_send, lpfn_send, false);
```

```
CPatch patch_for_recv(recv, my_recv, &lpfn_recv, false);
//so, patch is ready but is not applied. Call set_patch to apply it.
```

or:

```
CPatch patch_for_send(&lpfn_send, my_send, false);
CPatch patch_for_recv(lpfn_recv, my_recv, false);
```

If you would like to set the patch once forever, set the last parameter to `true`. This is a default parameter implicitly set to `false`.

```
//apply immediately, once forever:
CPatch patch_for_send(&lpfn_send, my_send, false, true);
//apply later, once forever:
CPatch patch_for_recv(lpfn_recv, my_recv, true, true);
```

Now the destructor will not remove patch. But you still can do it via function `remove_patch`.

The member function `patched()` tells you if path is set or not set. If you `set_patch`, then function returns `true`. If you remove patch, function returns `false`. Function `ok()` tells if patch generation could be created or not, and if it is possible to rewrite executable instructions or not. If `ok()` returns `false`, then `set_patch`/`remove_patch` will do nothing.
By the way, this approach does not affect the patch performance.

All other functions are `protected` or `private`. I don't think you will need to call them directly. But if you really need, you may change the code to make them `public`.

Intercepting Class Functions

Some time ago, I wanted to intercept class functions. After several changes I could do that with the patcher. Let's suppose, we have a class function we want to patch:

```
class aclass
{
public:
   void doSomething(int a, int b)
   {
      wcout<< L"void doSomething("<< a<<L", "<< b<< L")"<< endl;
   }
};
```

Because member functions are called in a special way, and a hidden parametter `this` is passed, we need to follow same logic. We will create a class, and will put the patch here. In this simple way, we will avoid all problems with calling in some bad way:

```
void (aclass::*pfn_doSomething)(int, int) = &aclass::doSomething;
class class_for_patches
{
public:

   void my_doSomething (int a, int b)
   {
      wcout<< L"patch void doSomething("<< a<< L","<< b<< L")"<< flush;
      a += b; //do anything
    b++;
      (reinterpret_cast<aclass*>(this)->*pfn_doSomething)(a, b);
   }
```

```
};
//.....
int main()
{
   //Let's see how it works
 CPatch patch_aclass_doSomething
       ( pfn_doSomething,       &bclass::my_doSomething );
   aclass x;
   x.doSomething(123, 234);
   //...
```

For these purposes, I have added several new constructors. You may take a look in *patcher_defines.h* inside *CPatcher.zip*.

How It Works

If you would like to see how the patching is applied/removed, take a look at the realisation of functions `CPatch::set_patch`/`CPatch::remove_patch`. These functions replace the start executable instructions of the function, and replace them with a `jmp`to the hook function. I tried to make them as simple as possible.

The most part of surgery is done in `CPatch::HookFunction(...)`. This function generates executable instructions for trampoline, computes offsets for `jmp`instructions and calls `set_patch`if the object intends to set patch immediately. It is called only once inside the constructor. One more thing about `jmp`instruction that you have to know, the address for `jmp`is calculated as relative to address of first byte after end of `jmp`instruction. More shortly, it is relative to `address of jmp + 5`.

Function `okToRewriteTragetInstructionSet`parses the instruction set from patched function startup. It returns the smallest number `N` of bytes needed to rewrite. This `N` must not be smaller than 5 bytes, because `jmp`instruction needs 5 bytes. The trampoline function contains exactly `N` + 5 bytes, that are filled with the first `N` bytes from the startup of patched function, and a `jmp`to the next instruction of patched function. So you can call the trampoline instead of original patched function. I think what `okToRewriteTragetInstructionSet`may be incomplete. I filled only instructions that I've found when debugging. For most `WinAPI`functions, this should be enough. But if it is not, you may add some new instructions in `okToRewriteTragetInstructionSet`. All you need, is to copy/paste from debugger.

For security reasons, the trampoline functions are allocated in a separate heap. We have to set some of the `PAGE_EXECUTE*` attributes. Otherwise the trampoline will not work. If we request this attribute via `VirtualProtect`, the attributes are set to whole heap. And changing attributes on default heaps to `PAGE_EXECUTE*` may lead to different types of flaws.

This patcher can be used not only inside the current application, but you may create a DLL and inject it into some process.

To inject a DLL in some process is quite simple.
First of all, you need to open the process. You may find the ID of the process in the Windows task manager. You also may find the `ProcessID`by using `EnumProcesses`, open each process, check the name with `GetModuleBaseName`.

If you used `CreateProcess` to start you process, then you already have the process `ID` in the returned `PROCESS_INFORMATION`. But in this case, you don't need it. You already have `hProcess` member of `PROCESS_INFORMATION`. Since you have the `hProcess`, you don't need to call `OpenProcess` now.

Since you got the process ID (`PID`), you open the process, alloc a buffer inside that process, and call `LoadLibrary` into that process, by using `CreateRemoteThread`, and voila:

```
HANDLE hProcess = OpenProcess(
       PROCESS_CREATE_THREAD | PROCESS_QUERY_INFORMATION |
        PROCESS_VM_OPERATION |PROCESS_VM_WRITE | PROCESS_VM_READ |
        PROCESS_TERMINATE , FALSE, PID);
if(hProcess)
```

```
{
   #define INJECTED_DLL L"InjectDllTest.dll"


   DWORD dwThreadId;
   LPVOID pvProcMem = VirtualAllocEx(hProcess, 0, sizeof(INJECTED_DLL),
                                     MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
   SIZE_T written;
   WriteProcessMemory(hProcess, pvProcMem, INJECTED_DLL, sizeof(INJECTED_DLL), &written);


   DWORD oldProtectRemoteThreadProc;
   HANDLE hThread = CreateRemoteThread(hProcess, 0, 0,
                                      (LPTHREAD_START_ROUTINE)LoadLibraryW,
                                      pvProcMem, 0, &dwThreadId);
   DWORD wtso = WAIT_FAILED;
   wtso = WaitForSingleObject(hThread, 1000);
   //check wtso to be sure what you code is successfully injected
```

Please, don't use `#define`and `sizeof`to pass the name of DLL you want to inject. This code is just intended to be as simple as possible.

If you need to set a global patch on the operation system, then you have to rewrite function entry of the DLL export table containing that function.

To be continued...

License

About the Author

armagedescu

Software Developer
Thomson Reuters
Moldova (Republic Of)

Member

I'm C/C++ developer since 1999.
My favorite language is C++ and my favorite programming environment is Microsoft Visual Studio.