QuietRIATT: Rebuilding the Import Address Table Using Hooked DLL Calls

Jason Raber (jraber@rri-usa.org)
Brian Krumheuer (bkrumheuer@rri-usa.org)

Riverside Research Institute

January 2009

Abstract

For a Reverse Engineer, rebuilding a large Import Address Table (IAT) can be a very time-consuming and tedious process. When the IAT has been sufficiently hashed or munged and current IAT rebuilders fail to resolve any of the calls, there is little other choice than to rebuild it by hand. Depending on the size, it can take days or even weeks. Also, doing anything by hand is prone to mistakes. QuietRIATT is an IDA Pro plug-in which automates the process of rebuilding the IAT when it can't be done by current IAT tools. Not only can it greatly reduce the amount of time spent rebuilding by hand, it also

removes the element of human error.

Details

Unlike the few tools currently available for IAT rebuilding, QuietRIATT does not simply try to read the IAT from the running program. It even goes beyond the tracing methods included in those tools. What makes QuietRIATT different is that it uses hooked Dll (e.g. Kernel32.dll, user32.dll) calls to generate a list of the return address and name of each DLL call that is used in a run of the program. This list is then used to create a new IAT. This method can be used to rebuild the IAT when other tools fail. Unless the malware detects the tracing of the DLL calls and causes the program to stop running.

QuietRIATT was designed for use when attempting to reverse engineer a piece of malware, that successfully munges or redirects the IAT to prevent ImpREC from rebuilding the IAT. Usually, tools like ImpREC with using the auto-trace feature will be able to fix it automatically, but if the IAT is re-directed enough times, or if the protection uses a hash function to resolve the external addresses, those tools won't be able to do the job. QuietRIATT was created to automate cases that those tools couldn't handle.

QuietRIATT is a three step process.

1) A modified version of Microsoft's Detours is used to hook DLL calls. When this is attached to the program, it records when DLL calls are made. A text file is generated with the names and return addresses of the calls used.
2) QuietRIATT parses this text file and annotates an IDAPro database with the information. From this database, it creates another text file containing the details of the IAT.
3) This new file is loaded into ImpREC which adds the new IAT to a dump of the original program.

Detours

Microsoft Detours is used to inject jumps into the functions contained in DLLs that get loaded with a particular program. When one of these functions is called, execution immediately jumps to our "hooked" function where we are able to print out information about the call, including the function name and return address. From here, we have the option of calling the "real" function. When it is finished, execution is sent back to our hooked code, which then returns to the normal program (Figure 1). All of this is done without modifying the code of the original program.

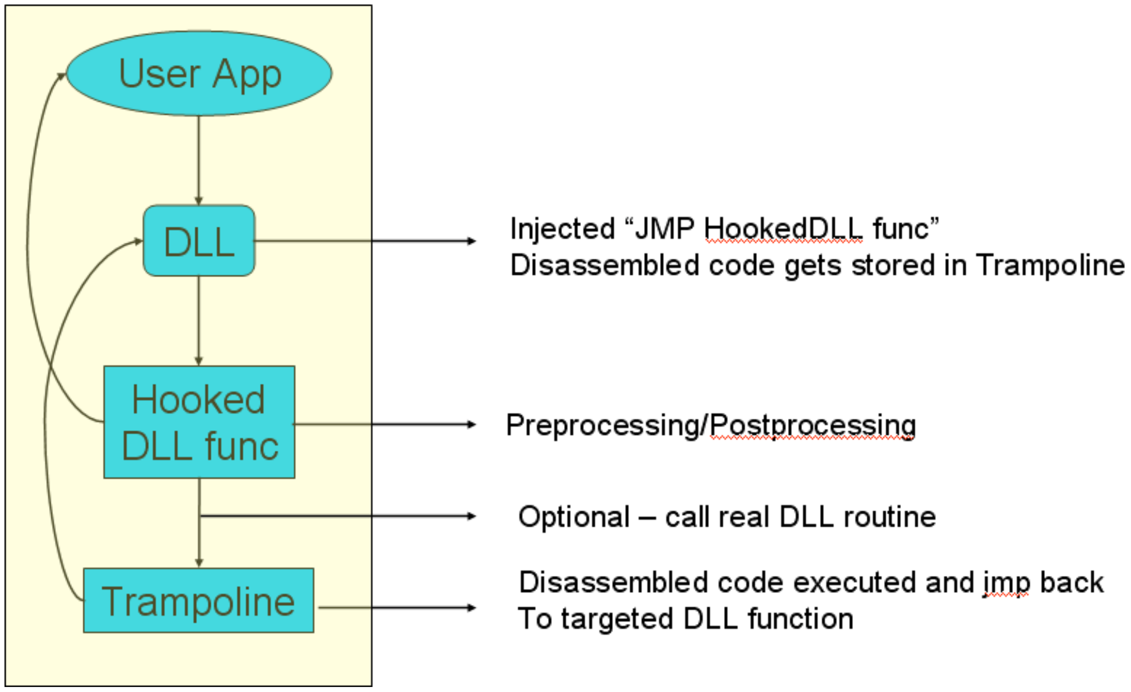

User App
DLL
Hooked DLL func
Trampoline
Injected "JMP HookedDLL func"
Disassembled code gets stored in Trampoline
Preprocessing/Postprocessing
Optional – call real DLL routine
Disassembled code executed and jmp back
To targeted DLL function

Figure 1: The Detours Function-Hooking Method

To set up a hooked call in detours, it is necessary to first create a pointer to the original function (Figure 2).

```
void (__stdcall * Real_GetSystemTimeAsFileTime)(LPFILETIME a0)
    = GetSystemTimeAsFileTime;
```

Figure 2: Pointer to Original DLL Function

Next, make the hooked function which prints some information about the function (Figure 3).

```
void __stdcall Mine_GetSystemTimeAsFileTime(LPFILETIME a0)
{
    PRINT_CALLER;
    _PrintEnter("GetSystemTimeAsFileTime (%p)\n", a0);

    __try {
        Real_GetSystemTimeAsFileTime(a0);
    } __finally {
        _PrintExit("GetSystemTimeAsFileTime  () ->\n");
    };
}
```

Figure 3: Hooked Function

Detours does not include the ability to print the return address, so some code needs to be added for QuietRIATT to do its job. In each hooked function, it is necessary to inject a small macro that reads the return address off the stack and prints it (Figure 4).

```
#define GET_CALLER_ADDR  \
{                        \
   __asm mov eax, ebp    \
   __asm add eax, 4      \
   __asm mov pStack, eax \
}

#define PRINT_CALLER \
{                    \
   int *pStack = 0;  \
   GET_CALLER_ADDR   \
   _Print("[[[ %X ]]]\n", *pStack ); \
}
```
Figure 4: Macro to Print Return Address

The last step is to actually hook the call using the function DetourAttach (Figure 5).

```
DetourAttach(&(PVOID&)Real_GetSystemTimeAsFileTime,
             &(PVOID&)Mine_GetSystemTimeAsFileTime);
```
Figure 5: Hooking the Function

When this modified Detours is attached to a program and executed, it will generate an output file containing the details of every hooked DLL function call that was made during the run of the program (Figure 6).

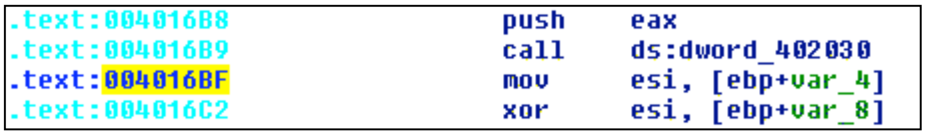
```
IF: 001 [[[ 4016BF ]]]
IF: 001 GetSystemTimeAsFileTime (13ffb4)
IF: 001 GetSystemTimeAsFileTime  () ->
IF: 001 [[[ 4016CB ]]]
IF: 001 GetCurrentProcessId ()
IF: 001 GetCurrentProcessId () -> db4
IF: 001 [[[ 4016D3 ]]]
IF: 001 GetCurrentThreadId ()
IF: 001 GetCurrentThreadId () -> d18
```
Figure 6: Output from Modified Detours

QuietRIATT

Internally, the QuietRIATT IDA Pro plug-in has two main loops. The first loop parses through the Detours output file and annotates the IDA Pro database with the IAT info. The second loop reads that info and creates the ImpREC tree file.

When the first loop finds a return address in the Detours output file (Figure 6) it goes to that address in the IDA Pro database (Figure 7). This address is the instruction immediately following the DLL call. In order to get the address of the call itself, the IDA

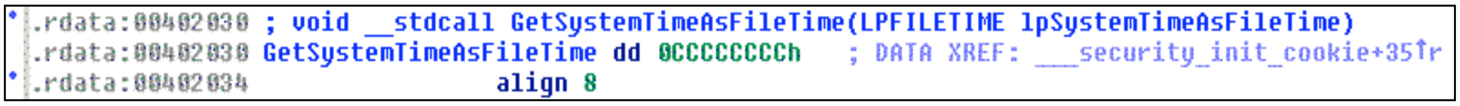
Pro SDK includes the function `decode_prev_insn(ea_t ea)` in the file ua.hpp, which returns the address of the previous instruction. In this example, the address of the call is 0x4016B9.



```
.text:004016B8                      push    eax
.text:004016B9                      call    ds:dword_402030
.text:004016BF                      mov     esi, [ebp+var_4]
.text:004016C2                      xor     esi, [ebp+var_8]
```

Figure 7: IDA Pro Disassembly

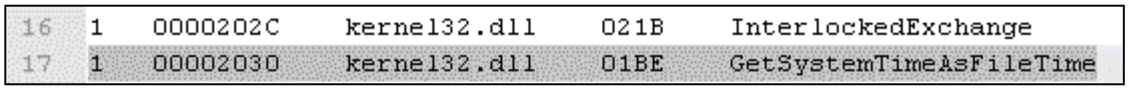
The next step is to analyze the call and extract the address of the IAT entry for the function. The IDA Pro SDK provides functionality for this, as well. First, `ua_ana0(ea_t ea)` is used to fill the global `cmd` variable with information about the instruction. The `cmd` variable now contains the address of the IAT entry, which can be accessed as the first operand of the instruction (`cmd.Operands[0].addr`). Finally, we go to that address and rename it with the name of the function from the Detours output (Figure 8).



```
.rdata:00402030 ; void __stdcall GetSystemTimeAsFileTime(LPFILETIME lpSystemTimeAsFileTime)
.rdata:00402030 GetSystemTimeAsFileTime dd 0CCCCCCCCh    ; DATA XREF: ___security_init_cookie+35↑r
.rdata:00402034                         align 8
```

Figure 8: Re-named IAT Entry in IDA Pro Database

After this has been done for every call in the Detours output, it is time for the second loop. This one goes through the annotated IAT in the IDA database and creates a file with the IAT Relative Virtual Address (RVA), DLL name, ordinal and function name for each entry (Figure 9). This is a “tree” file which is then imported into ImpREC.



```
16  1  0000202C   kernel32.dll   021B   InterlockedExchange
17  1  00002030   kernel32.dll   01BE   GetSystemTimeAsFileTime
```

Figure 9: ImpREC Tree File

The IAT RVA and function name are read directly from the IDA database, but the DLL name and ordinal are not available this way. To find these, it is necessary to disassemble the DLL beforehand and create a list of the functions and ordinals. The list is stored in the file QuietRIATT_liblist.txt which must be in the IDA\plugins directory. All DLL functions that a program uses must be included in this list which is read into QuietRIATT at startup. When a function is encountered, QuietRIATT searches this list for the function name, retrieves the corresponding information, and writes it to the tree file.

To load the tree file into ImpREC, it is first necessary to run the application that is being rebuilt and attach ImpREC to the running process. Then, click the “Load Tree” button in the bottom left corner and choose the tree file created by QuietRIATT. Lastly, click the “Fix Dump” button, choose the program’s executable file and ImpREC will take care of the rest. It will create a new file with the same name as the original followed by an underscore.

Special Cases

The process described above is a best-case scenario. Most often things are not so easy, and some additional work is necessary. Sometimes the return address read from the Detours output file turns out to be in a section of code that IDA has not disassembled. When this happens, the function for decoding the previous instruction is pretty much useless. In order to find the address of the call instruction, QuietRIATT will analyze the bytes immediately before the return address to see if it can find the call. If it finds a call, it continues and tries to find the address being called. If not, it moves on to the next return address in the Detours output file.

Not all calls that are made are in the form: CALL <memory addr>. Sometimes they are indirect calls where the address is stored in a register, like: CALL EAX. It is not always possible to find out how the address got into the register, but QuietRIATT will check up to 32 instructions preceding the call to see if it can be found. In the case where the address can not be found, QuietRIATT will display a warning in the IDA message window with the address of the call instruction and the name of the function it called. When the warning is double-clicked, the disassembly will jump to that address so the user can attempt to fix it manually.

If there is some redirection of the IAT (for example, the IAT addresses are copied to different locations in memory, and those addresses are used when calling DLL functions), QuietRIATT will attempt to get past it. If QuietRIATT determines that a DLL function is being called, but the address being called is in a memory location outside of the IAT, it will check the cross-references (xrefs) of that memory location. For each xref, it will see if there is a MOV instruction where data is being copied into the memory location. The source will usually be a register since direct memory to memory transfers are not possible. The next step is to search the instructions preceding the MOV to see if there is data being put in the same register that is being used in the MOV. Hopefully, this data is an entry in the IAT. If so, QuietRIATT will rename the IAT entry with the name of the function that was called.

Since Detours will only record DLL calls that were made during a run of the program, and not all of the functions in the IAT will be used in every run, there will often be functions in the IAT that QuietRIATT will not be able to resolve. Since there has to be something in every IAT entry, a default function will have to be chosen for every DLL. This function will be used in place of those that are not known. In order to get the highest number of identified DLL functions, it is recommended that when running the program with Detours attached, try to use the program to its fullest extent. It is possible that even when the program is run with every available option, not all of the DLL functions are identified. However, usually enough functions will be identified to get the program in a runnable condition.

Bios

Jason Raber
Jason serves as the technical lead for the Riverside Research Institute Red Team which provides government and commercial entities with specialized software security support.

Focus areas include:

- **Reverse Engineering**: Specializes in extracting intellectual property from a broad spectrum of software. This includes user applications, DLLs, drivers, OS kernels, and firmware. The software can be based on a variety of platforms (Windows/Linux/Mac/Embedded, etc.).

- **Malware/Virus/RootKit Analysis**: Identifies and analyzes intrusion software to characterize and/or neutralize the threat.

Jason has spent seven years in the world of reverse engineering, preceded by five years working at Texas Instruments developing Compiler tools for DSPs (code generators, assemblers, linkers, disassemblers, etc.). Developing C compilers for five years prior to reverse engineering has provided a good foundation for understanding machine language and hardware to be utilized in reverse engineering tasks.

Brian Krumheuer
Brian is a Reverse Engineer on the Software Security Team at Riverside Research Institute. He worked for over eight years in IT and Software Development before entering the field of Reverse Engineering. Currently, he plays a vital role on the team by developing many ring-3 and ring-0 reverse engineering tools for both Windows and Linux. He has also helped create an instructional course on Reverse Engineering.