Improving Automated Analysis of Windows x64 Binaries

April, 2006

skape
mmiller@hick.org

Contents

1 Foreword **2**

2 Introduction **3**

3 Background **4**

3.1 PE32+ Image File Format 4

3.2 Calling Convention 5

3.2.1 Stack Frame Layout 5

3.3 Exception Handling on x64 7

3.3.1 Exception Directory 8

3.3.2 Unwind Information 8

4 Analysis Techniques **11**

4.1 Exception Directory Enumeration 11

4.1.1 Functions 11

4.1.2 Stack Frame Annotation 12

4.1.3 Exception Handlers 15

4.2 Register Parameter Area Annotation 16

5 Conclusion **17**

Chapter 1

Foreword

Abstract: As Windows x64 becomes a more prominent platform, it will become necessary to develop techniques that improve the binary analysis process. In particular, automated techniques that can be performed prior to doing code or data flow analysis can be useful in getting a better understanding for how a binary operates. To that point, this paper gives a brief explanation of some of the changes that have been made to support Windows x64 binaries. From there, a few basic techniques are illustrated that can be used to improve the process of identifying functions, annotating their stack frames, and describing their exception handler relationships. Source code to an example IDA plugin is also included that shows how these techniques can be implemented.

Thanks: The author would like to thank bugcheck, sh0k, jt, spoonm, and Skywing.

Update: The article in MSDN magazine by Matt Pietrek was published after this article was written. However, it contains a lot of useful information and touches on many of the same topics that this article covers in the background chapter. The article can be found here: `http://msdn.microsoft.com/msdnmag/issues/06/05/x64/default.aspx`.

With that, on with the show…

Chapter 2

Introduction

The demand for techniques that can be used to improve the analysis process of Windows x64 binaries will only increase as the Windows x64 platform becomes more accepted and used in the market place. There is a deluge of useful information surrounding techniques that can be used to perform code and data flow analysis that is also applicable to the x64 architecture. However, techniques that can be used to better annotate and streamline the initial analysis phases, such as identifying functions and describing their stack frames, is still a ripe area for improvement at the time of this writing. For that reason, this paper will start by describing some of the changes that have been made to support Windows x64 binaries. This background information is useful because it serves as a basis for understanding a few basic techniques that may be used to improve some of the initial analysis phases. During the course of this paper, the term *Windows x64 binary* will simply be reduced to *x64 binary* in the interest of brevity.

Chapter 3

Background

Prior to diving into some of the analysis techniques that can be performed on x64 binaries, it's first necessary to learn a bit about some of the changes that were made to support the x64 architecture. This chapter will give a very brief explanation of some of the things that have been introduced, but will by no means attempt to act as an authoritative reference.

3.1 PE32+ Image File Format

The image file format for the x64 platform is known as PE32+. As one would expect, the file format is derived from the PE file format with only very slight modifications. For instance, 64-bit binaries contain an IMAGE_OPTIONAL_HEADER64 rather than an IMAGE_OPTIONAL_HEADER. The differences between these two structures are described in the table below:

Field	PE	PE32+
BaseOfData	ULONG	Removed from structure
ImageBase	ULONG	ULONGLONG
SizeOfStackReserve	ULONG	ULONGLONG
SizeOfStackCommit	ULONG	ULONGLONG
SizeOfHeapReserve	ULONG	ULONGLONG
SizeOfHeapCommit	ULONG	ULONGLONG

Figure 3.1: IMAGE_OPTIONAL_HEADER differences

In general, any structure attribute in the PE image that made reference to a 32-bit virtual address directly rather than through an RVA (Relative Virtual Address) has been expanded to a 64-bit attribute in PE32+. Other examples of this include the IMAGE_TLS_DIRECTORY structure and the IMAGE_LOAD_CONFIG_DIRECTORY structure.

With the exception of certain field offsets in specific structures, the PE32+ image file format is largely backward compatible with PE both in use and in form.

3.2 Calling Convention

The calling convention used on x64 is much simpler than those used for x86. Unlike x86, where calling conventions like `stdcall`, `cdecl`, and `fastcall` are found, the x64 platform has only one calling convention. The calling convention that it uses is a derivative of `fastcall` where the first four parameters of a function are passed by register and any remaining parameters are passed through the stack. Each parameter is 64 bits wide (8 bytes). The first four parameters are passed through the RCX, RDX, R8, and R9 registers, respectively. For scenarios where parameters are passed by value or are otherwise too large to fit into one of the 64-bit registers, appropriate steps are taken as documented in [4].

3.2.1 Stack Frame Layout

The stack frame layout for functions on x64 is very similar to x86, but with a few key differences. Just like x86, the stack frame on x64 is divided into three parts: parameters, return address, and locals. These three parts are explained individually below. One of the important principals to understand when it comes to x64 stack frames is that the stack does not fluctuate throughout the course of a given function. In fact, the stack pointer is only permitted to change in the context of a function prologue[1]. Parameters are not pushed and popped from the stack. Instead, stack space is pre-allocated for all of the arguments that would be passed to child functions. This is done, in part, for making it easier to unwind call stacks in the event of an exception. The table below describes a typical stack frame:

Stack parameter area
Register parameter area
Return address
Locals

[1] Note that things like `alloca` are handled in a special manner[7]

Parameters

The calling convention for functions on x64 dictates that the first four parameters are passed via register with any remaining parameters, starting with parameter five, spilling to the stack. Given that the fifth parameter is the first parameter passed by the stack, one would think that the fifth parameter would be the value immediately adjacent to the return address on the stack, but this is not the case. Instead, if a given function calls other functions, that function is required to allocate stack space for the parameters that are passed by register. This has the affect of making it such that the area of the stack immediately adjacent to the return address is `0x20` bytes of uninitialized storage for the parameters passed by register followed immediately by any parameters that spill to the stack (starting with parameter five). The area of storage allocated on the stack for the register parameters is known as the *register parameter area* whereas the area of the stack for parameters that spill onto the stack is known as the *stack parameter area*. The table below illustrates what the parameter portion of a stack frame would look like after making a call to a function:

Parameter 6
Parameter 5
Parameter 4 (`R9` Home)
Parameter 3 (`R8` Home)
Parameter 2 (`RDX` Home)
Parameter 1 (`RCX` Home)
Return address

To emphasize further, the register parameter area is always allocated, even if the function being called has fewer than four arguments. This area of the stack is effectively owned by the called function, and as such can be used for volatile storage during the course of the function call. In particular, this area is commonly used to persist the values of register parameters[2]. However, it can also be used to save non-volatile registers. To someone familiar with x86 it may seem slightly odd to see functions modifying areas of the stack beyond the return address. The key is to remember that the `0x20` bytes immediately adjacent to the return address are owned by the called function. One important side affect of this requirement is that if a function calls other functions, the calling function's minimum stack allocation will be `0x20` bytes. This accounts for the register parameter area that will be used by called functions.

The obvious question to ask at this point is why it's the caller's responsibility to allocate stack space for use by the called function. There are a few different reasons for this. Perhaps most importantly, it makes it possible for the called function to take the address of a parameter that's passed via a register. Furthermore, the address that is returned for the parameter must be at a location that

[2] This area is also referred to as the "home" address for register parameters

is contiguous in relation to the other parameters. This is particularly necessary for variadic functions, which require a contiguous list of parameters, but may also be necessary for applications that make assumptions about being able to reference parameters in relation to one another by address. Invalidating this assumption would introduce source compatibility problems.

For more information on parameter passing, refer to the MSDN documentation[4, 7].

Return Address

Due to the fact that pointers are 64 bits wide on x64, the return address location on the stack is eight bytes instead of four.

Locals

The locals portion of a function's stack frame encompasses both local variables and saved non-volatile registers. For x64, the general purpose registers described as non-volatile are `RBP`, `RBX`, `RDI`, `RSI`, and `R12` through `R15`[5].

3.3 Exception Handling on x64

On x86, exception handling is accomplished through the adding and removing of exception registration records on a per-thread basis. When a function is entered that makes use of an exception handler, it constructs an exception registration record on the stack that is composed of an exception handler (a function pointer), and a pointer to the next element in the exception handler list. This list of exception registration records is stored relative to `fs:[0]`. When an exception occurs, the exception dispatcher walks the list of exception handlers and calls each one, checking to see if they are capable of handling the exception that occurred. While this approach works perfectly fine, Microsoft realized that there were better ways to go about it. First of all, the adding and removing of exception registration records that are static in the context of an execution path adds needless execution overhead. Secondly, the security implications of storing a function pointer on the stack have been made very obvious, especially in the case where that function pointer can be called after an exception is generated (such as an access violation). Finally, the process of unwinding call frames is muddled with limitations, thus making it a more complicated process than it might otherwise need to be[6].

With these things in mind, Microsoft completely revamped the way exception handling is accomplished on x64. The major changes center around the approaches Microsoft has taken to solve the three major deficiencies found on x86. First, Microsoft solved the execution time overhead issue of adding and removing exception handlers by moving all of the static exception handling information into a static location in the binary. This location, known as the `.pdata` section, is described by the PE32+'s `Exception Directory`. The structure of this section will be described in the *exception directory* subsection. By eliminating the need to add and remove exception handlers on the fly, Microsoft has also eliminated the security issue found on x86 with regard to overwriting the function pointer of an exception handler. Perhaps most importantly, the process involved in unwinding call frames has been drastically improved through the formalization of the frame unwinding process. This will be discussed in the subsection on *unwind information.*

3.3.1 Exception Directory

The `Exception Directory` of a PE32+ binary is used to convey the complete list of functions that could be found in a stack frame during an unwind operation. These functions are known as non-leaf functions, and they are qualified as such if they either allocate space on the stack or call other functions. The `IMAGE_RUNTIME_FUNCTION_ENTRY` data structure is used to describe the non-leaf functions, as shown below[1]:

```
typedef struct _IMAGE_RUNTIME_FUNCTION_ENTRY {
    ULONG BeginAddress;
    ULONG EndAddress;
    ULONG UnwindInfoAddress;
} _IMAGE_RUNTIME_FUNCTION_ENTRY, *_PIMAGE_RUNTIME_FUNCTION_ENTRY;
```

The `BeginAddress` and `EndAddress` attributes are RVAs that represent the range of the non-leaf function. The `UnwindInfoAddress` will be discussed in more detail in the following subsection on unwind information. The Exception directory itself is merely an array of `IMAGE_RUNTIME_FUNCTION_ENTRY` structures. When an exception occurs, the exception dispatcher will enumerate the array of runtime function entries until it finds the non-leaf function associated with the address it's searching for (typically a return address).

3.3.2 Unwind Information

For the purpose of unwinding call frames and dispatching exceptions, each non-leaf function has some non-zero amount of unwind information associated with it. This association is made through the `UnwindInfoAddress` attribute of the `IMAGE_RUNTIME_FUNCTION_ENTRY` structure. The `UnwindInfoAddress` itself is an RVA that points to an `UNWIND_INFO` structure which is defined as[8]:

```
typedef struct _UNWIND_INFO {
    UBYTE Version       : 3;
    UBYTE Flags         : 5;
    UBYTE SizeOfProlog;
    UBYTE CountOfCodes;
    UBYTE FrameRegister : 4;
    UBYTE FrameOffset   : 4;
    UNWIND_CODE UnwindCode[1];
/*  UNWIND_CODE MoreUnwindCode[((CountOfCodes + 1) & ~1) - 1];
*   union {
*       OPTIONAL ULONG ExceptionHandler;
*       OPTIONAL ULONG FunctionEntry;
*   };
*   OPTIONAL ULONG ExceptionData[]; */
} UNWIND_INFO, *PUNWIND_INFO;
```

This structure, at a very high level, describes a non-leaf function in terms of its prologue size and frame register usage. Furthermore, it describes the way in which the stack is set up when the prologue for this non-leaf function is executed. This is provided through an array of codes as accessed through the `UnwindCode` array. This array is composed of `UNWIND_CODE` structures which are defined as[8]:

```
typedef union _UNWIND_CODE {
    struct {
        UBYTE CodeOffset;
        UBYTE UnwindOp : 4;
        UBYTE OpInfo   : 4;
    };
    USHORT FrameOffset;
} UNWIND_CODE, *PUNWIND_CODE;
```

In order to properly unwind a frame, the exception dispatcher needs to be aware of the amount of stack space allocated in that frame, the locations of saved non-volatile registers, and anything else that has to do with the stack. This information is necessary in order to be able to restore the caller's stack frame when an unwind operation occurs. By having the compiler keep track of this information at link time, it's possible to emulate the unwind process by inverting the operations described in the unwind code array for a given non-leaf function.

Aside from conveying stack frame set up, the `UNWIND_INFO` structure may also describe exception handling information, such as the exception handler that is to be called if an exception occurs. This information is conveyed through the `ExceptionHandler` and `ExceptionData` attributes of the structure which exist only if the `UNW_FLAG_EHANDLER` flag is set in the `Flags` field.

For more details on the format and use of these structures for unwinding as well as a complete description of the unwind process, please refer to the MSDN documentation[2].

Chapter 4

Analysis Techniques

In order to improve the analysis of x64 binaries, it is important to try to identify techniques that can aide in the identification or extraction of useful information from the binary in an automated fashion. This chapter will focus on a handful of simple techniques that can be used to better annotate or describe the behavior of an x64 binary. These techniques intentionally do not cover the analysis of code or data flow operations. Such techniques are outside of the scope of this paper.

4.1 Exception Directory Enumeration

Given the explanation of the `Exception Directory` found within PE32+ images and its application to the exception dispatching process, it can be seen that x64 binaries have a lot of useful meta-information stored within them. Given that this information is just sitting there waiting to be used, it makes sense to try to take advantage of it in ways that make it possible to better annotate or understand an x64 binary. The following subsections will describe different things that can be discovered by digging deeper into the contents of the exception directory.

4.1.1 Functions

One of the most obvious uses for the information stored in the exception directory is that it can be used to discover all of the non-leaf functions in a binary. This is cool because it works regardless of whether or not you actually have symbols for the binary, thus providing an easy technique for identifying the majority of the functions in a binary. The process taken to do this is to simply enumerate the array of IMAGE_RUNTIME_FUNCTION_ENTRY structures stored within the exception directory. The BeginAddress attribute of each entry marks the starting point of a non-leaf function. There's a catch, though. Not all of the runtime function entries are actually associated with the entry point of a function. The fact of the matter is that entries can also be associated with various portions of an actual function where stack modifications are deferred until necessary. In these cases, the unwind information associated with the runtime function entry is chained with another runtime function entry.

The chaining of runtime function entries is documented as being indicated through the UNW_FLAG_CHAININFO flag in the Flags attribute of the UNWIND_INFO structure. If this flag is set, the area of memory immediately following the last UNWIND_CODE in the UNWIND_INFO structure is an IMAGE_RUNTIME_FUNCTION_ENTRY structure. The UnwindInfoAddress of this structure indicates the chained unwind information. Aside from this, chaining can also be indicated through an undocumented flag that is stored in the least-significant bit of the UnwindInfoAddress. If the least-significant bit is set, then it is implied that the runtime function entry is directly chained to the IMAGE_RUNTIME_FUNCTION_ENTRY structure that is found at the RVA conveyed by the UnwindInfoAddress attribute with the least significant bit masked off. The reason chaining can be indicated in this fashion is because it is a requirement that unwind information be four byte aligned.

With chaining in mind, it is safe to assume that a runtime function entry is associated with the entry point of a function if its unwind information is *not* chained. This makes it possible to deterministically identify the entry point of all of the non-leaf functions. From there, it should be possible to identify all of the leaf functions through calls that are made to them by non-leaf functions. This requires code flow analysis, though.

4.1.2 Stack Frame Annotation

The unwind information associated with each non-leaf function contains lots of useful meta-information about the structure of the stack. It provides information about the amount of stack space allocated, the location of saved non-volatile registers, and whether or not a frame register is used and what relation it has to the rest of the stack. This information is also described in terms of the location of the instruction that actually performs the operation associated with the task. Take the following unwind information obtained through dumpbin /unwindinfo as an example:

```
0000060C 00006E50 00006FF0 000081FC  _resetstkoflw
  Unwind version: 1
  Unwind flags: None
  Size of prologue: 0x47
  Count of codes: 18
```

```
Frame register: rbp
Frame offset: 0x20
Unwind codes:
  3C: SAVE_NONVOL, register=r15 offset=0x98
  38: SAVE_NONVOL, register=r14 offset=0xA0
  31: SAVE_NONVOL, register=r13 offset=0xA8
  2A: SAVE_NONVOL, register=r12 offset=0xD8
  23: SAVE_NONVOL, register=rdi offset=0xD0
  1C: SAVE_NONVOL, register=rsi offset=0xC8
  15: SAVE_NONVOL, register=rbx offset=0xC0
  0E: SET_FPREG, register=rbp, offset=0x20
  09: ALLOC_LARGE, size=0xB0
  02: PUSH_NONVOL, register=rbp
```

First and foremost, one can immediately see that the size of the prologue used in the _resetstkoflw function is 0x47 bytes. This prologue accounts for all of the operations described in the unwind codes array. Furthermore, one can also tell that the function uses a frame pointer, as conveyed through rbp, and that the frame pointer offset is 0x20 bytes relative to the current stack pointer at the time the frame pointer register is established.

As one would expect with an unwind operation, the unwind codes themselves are stored in the opposite order of which they are executed. This is necessary because of the effect on the stack each unwind code can have. If they are processed in the wrong order, then the unwind operation will get invalid data. For example, the value obtained through a pop rbp instruction will differ depending on whether or not it is done before or after an add rsp, 0xb0.

For the purposes of annotation, however, the important thing to keep in mind is how all of the useful information can be extracted. In this case, it is possible to take all of the information the unwind codes provide and break it down into a definition of the stack frame layout for a function. This can be accomplished by processing the unwind codes in the order that they would be executed rather than the order that they appear in the array. There's one important thing to keep in mind when doing this. Since unwind information can be chained, it is a requirement that the full chain of unwind codes be processed in execution order. This can be accomplished by walking the chain of unwind information and building an execution order list of all of the unwind codes.

Once the execution order list of unwind codes is collected, the next step is to simply enumerate each code, checking to see what operation it performs and building out the stack frame across each iteration. Prior to enumerating each code, the state of the stack pointer should be initialized to 0 to indicate an empty stack frame. As data is allocated on the stack, the stack pointer should be adjusted by the appropriate amount. The actions that need to be taken for each unwind operation that directly effect the stack pointer are described below.

1. **UWOP_PUSH_NONVOL**

   When a non-volatile register is pushed onto the stack, such as through a `push rbp`, the current stack pointer needs to be decremented by 8 bytes.

2. **UWOP_ALLOC_LARGE and UWOP_ALLOC_SMALL**

   When stack space is allocated, the current stack pointer needs to be adjusted by the amount indicated.

3. **UWOP_SET_FPREG**

   When a frame pointer is defined, its offset relative to the base of the stack should be saved using the current value of the stack pointer.

As the enumeration unwind codes occurs, it is also possible to annotate the different locations on the stack where non-volatile registers are preserved. For instance, given the example unwind information above, it is known that the `R15` register is preserved at `[rsp + 0x98]`. Therefore, we can annotate this location as `[rsp + SavedR15]`.

Beyond annotating preserved register locations on the stack, we can also annotate the instructions that perform operations that effect the stack. For instance, when a non-volatile register is pushed, such as through `push rbp`, we can annotate the instruction that performs that operation as preserving `rbp` on the stack. The location of the instruction that's associated with the operation can be determined by taking the `BeginAddress` associated with the unwind information and adding it to the `CodeOffset` attribute of the `UNWIND_CODE` that is being processed. It is important to note, however, that the `CodeOffset` attribute actually points to the first byte of the instruction immediately following the one that performs the actual operation, so it is necessary to back track in order to determine the start of the instruction that actually performs the operation.

As a result of this analysis, one can take the prologue of the `_resetstkoflw` function and automatically convert it from:

```
.text:100006E50     push rbp
.text:100006E52     sub rsp, 0B0h
.text:100006E59     lea rbp, [rsp+0B0h+var_90]
.text:100006E5E     mov [rbp+0A0h], rbx
.text:100006E65     mov [rbp+0A8h], rsi
.text:100006E6C     mov [rbp+0B0h], rdi
.text:100006E73     mov [rbp+0B8h], r12
.text:100006E7A     mov [rbp+88h], r13
.text:100006E81     mov [rbp+80h], r14
.text:100006E88     mov [rbp+78h], r15
```

to a version with better annotation:

```
.text:100006E50     push rbp                        ; SavedRBP
.text:100006E52     sub rsp, 0B0h
.text:100006E59     lea rbp, [rsp+20h]
.text:100006E5E     mov [rbp+0A0h], rbx             ; SavedRBX
.text:100006E65     mov [rbp+98h+SavedRSI], rsi    ; SavedRSI
.text:100006E6C     mov [rbp+98h+SavedRDI], rdi    ; SavedRDI
.text:100006E73     mov [rbp+98h+SavedR12], r12    ; SavedR12
.text:100006E7A     mov [rbp+98h+SavedR13], r13    ; SavedR13
.text:100006E81     mov [rbp+98h+SavedR14], r14    ; SavedR14
.text:100006E88     mov [rbp+98h+SavedR15], r15    ; SavedR15
```

While such annotation may is not entirely useful to understanding the behavior of the binary, it at least simplifies the process of understanding the layout of the stack.

4.1.3 Exception Handlers

The unwind information structure for a non-leaf function also contains useful information about the way in which exceptions within that function should be dispatched. If the unwind information associated with a function has the UNW_FLAG_EHANDLER or UNW_FLAG_UHANDLER flag set, then the function has an exception handler associated with it. The exception handler is conveyed through the ExceptionHandler attribute which comes immediately after the array of unwind codes. This handler is defined as being a language-specific handler for processing the exception. More specifically, the exception handler is specific to the semantics associated with a given programming language, such as C or C++[3]. For C, the language-specific exception handler is named __C_specific_handler.

Given that all C functions that handle exceptions will have the same exception handler, how does the function-specific code for handling an exception actually get called? For the case of C functions, the function-specific exception handler is stored in a scope table in the ExceptionData portion of the UNWIND_INFO structure[1]. This C scope table is defined by the structures shown below:

```
typedef struct _C_SCOPE_TABLE_ENTRY {
    ULONG Begin;
    ULONG End;
    ULONG Handler;
    ULONG Target;
} C_SCOPE_TABLE_ENTRY, *PC_SCOPE_TABLE_ENTRY;

typedef struct _C_SCOPE_TABLE {
    ULONG NumEntries;
```

[1] Other languages may have a different ExceptionData definition

```
    C_SCOPE_TABLE_ENTRY Table[1];
} C_SCOPE_TABLE, *PC_SCOPE_TABLE;
```

The scope table entries describe the function-specific exception handlers in relation to the specific areas of the function that they apply to. Each of the attributes of the C_SCOPE_TABLE_ENTRY is expressed as an RVA. The Target attribute defines the location to transfer control to after the exception is handled.

The reason why all of the exception handler information is useful is because it makes it possible to annotate a function in terms of what exception handlers may be called during its execution. It also makes it possible to identify the exception handler functions that may otherwise not be found due to the fact that they are executed indirectly. For example, the function CcAcquireByteRangeForWrite in ntoskrnl.exe can be annotated in the following fashion:

```
.text:0000000000434520 ; Exception handler: __C_specific_handler
.text:0000000000434520 ; Language specific handler: sub_4C7F30
.text:0000000000434520
.text:0000000000434520 CcAcquireByteRangeForWrite proc near
```

4.2 Register Parameter Area Annotation

Given the requirement that the register parameter area be allocated on the stack in the context of a function that calls other functions, it is possible to statically annotate specific portions of the stack frame for a function as being the location of the caller's register parameter area. Furthermore, the location of a given function's register parameter area that is to be used by called functions can also be annotated.

The location of the register parameter area is always at a fixed location in a stack frame. Specifically, it immediately follows the return address on the stack. If annotations are added for CallerRCX at offset 0x8, CallerRDX at offset 0x10, CallerR8 at offset 0x18, and CallerR9 at offset 0x20, it is possible to get a better view of the stack frame for a given function. It also makes it easier to understand when and how this region of the stack is used by a function. For instance, the CcAcquireByteRangeForWrite function in ntoskrnl.exe makes use of this area to store the values of the first four parameters:

```
.text:0000000000434520    mov     [rsp+CallerR9], r9
.text:0000000000434525    mov     dword ptr [rsp+CallerR8], r8d
.text:000000000043452A    mov     [rsp+CallerRDX], rdx
.text:000000000043452F    mov     [rsp+CallerRCX], rcx
```

Chapter 5

Conclusion

This paper has presented a few basic approaches that can be used to extract useful information from an x64 binary for the purpose of analysis. By analyzing the unwind information associated with functions, it is possible to get a better understanding for how a function's stack frame is laid out. Furthermore, the unwind information makes it possible to describe the relationship between a function and its exception handler(s). Looking toward the future, x64 is likely to become the standard architecture given Microsoft's adoption of it as their primary architecture. With this in mind, coming up with techniques to better automate the binary analysis process will become more necessary.

Bibliography

[1] Microsoft Corporation. *ntimage.h*. 3790 DDK header files.

[2] Microsoft Corporation. *Exception Handling (x64)*. http://msdn2.microsoft.com/en-us/library/1eyas8tf(VS.80).aspx; accessed Apr 25, 2006.

[3] Microsoft Corporation. *The Language Specific Handler*. http://msdn2.microsoft.com/en-us/library/b6sf5kbd(VS.80).aspx; accessed Apr 25, 2006.

[4] Microsoft Corporation. *Parameter Passing*. http://msdn2.microsoft.com/en-us/library/zthk2dkh.aspx; accessed Apr 25, 2006.

[5] Microsoft Corporation. *Register Usage*. http://msdn2.microsoft.com/en-us/library/9z1stfyw(VS.80).aspx; accessed Apr 25, 2006.

[6] Microsoft Corporation. *SEH in x86 Environments*. http://msdn2.microsoft.com/en-US/library/ms253960.aspx; accessed Apr 25, 2006.

[7] Microsoft Corporation. *Stack Usage*. http://msdn2.microsoft.com/en-us/library/ew5tede7.aspx; accessed Apr 25, 2006.

[8] Microsoft Corporation. *Unwind Data Definitions in C*. http://msdn2.microsoft.com/en-us/library/ssa62fwe(VS.80).aspx; accessed Apr 25, 2006.