Memalyze: Dynamic Analysis of Memory Access Behavior in Software

4/2007

skape
mmiller@hick.org

Abstract

This paper describes strategies for dynamically analyzing an application's memory access behavior. These strategies make it possible to detect when a read or write is about to occur at a given location in memory while an application is executing. An application's memory access behavior can provide additional insight into its behavior. For example, it may be able to provide an idea of how data propagates throughout the address space. Three individual strategies which can be used to intercept memory accesses are described in this paper. Each strategy makes use of a unique method of intercepting memory accesses. These methods include the use of Dynamic Binary Instrumentation (DBI), x86 hardware paging features, and x86 segmentation features. A detailed description of the design and implementation of these strategies for 32-bit versions of Windows is given. Potential uses for these analysis techniques are described in detail.

1 Introduction

If software analysis had a holy grail, it would more than likely be centered around the ability to accurately model the data flow behavior of an application. After all, applications aren't really much more than sophisticated data processors that operate on varying sets of input to produce varying sets of output. Describing how an application behaves when it encounters these varying sets of input makes it possible to predict future behavior. Furthermore, it can provide insight into how the input could be altered to cause the application to behave differently. Given these benefits, it's only natural that a discipline exists that is devoted to the study of data flow analysis.

There are a two general approaches that can be taken to perform data flow analysis. The first approach is referred to as static analysis and it involves analyzing an application's source code or compiled binaries without actually executing the application. The second approach is dynamic analysis which, as one would expect, involves analyzing the data flow of an application as it executes. The two approaches both have common and unique benefits and no argument will be made in this paper as to which may be better or worse. Instead, this paper will focus on describing three strategies that may be used to assist in the process of dynamic data flow analysis.

The first strategy involves using Dynamic Binary Instrumentation (DBI) to rewrite the instruction stream of the executing application in a manner that makes it possible to intercept instructions that read from or write to memory. Two well-known examples of DBI implementations that the author is familiar with are DynamoRIO and Valgrind[4, 12]. The second strategy that will be discussed involves using the hardware paging features of the x86 and x64 architectures to trap and handle access to specific pages in memory. Finally, the third strategy makes use of the segmentation features included in the x86 architecture to trap memory accesses by making use of the null selector. Though these three strategies vary greatly, they all accomplish the same goal of being able to intercept memory accesses within an application as it executes.

The ability to intercept memory reads and writes during runtime can support research in additional areas relating to dynamic data flow analysis. For example, the ability to track what areas of code are reading from and writing to memory could make it possible to build a model for the data propagation behaviors of an application. Furthermore, it might be possible to show with what degree of code-level isolation different areas of memory are accessed. Indeed, it may also be possible to attempt to validate the data consistency model of a threaded application by investigating the access behaviors of various regions of memory which are referenced by multiple threads. These are but a few of the many potential candidates for dynamic data flow analysis.

This paper is organized into three sections. Section 2 gives an introduction to three different strategies for facilitating dynamic data flow analysis. Section 3 enumerates some of the potential scenarios in which

these strategies could be applied in order to render some useful information about the data flow behavior of an application. Finally, section 4 describes some of the previous work whose concepts have been used as the basis for the research described herein.

2 Strategies

This section describes three strategies that can be used to intercept runtime memory accesses. The strategies described herein do not rely on any static binary analysis. Techniques that do make use of static binary analysis are outside of the scope of this paper.

2.1 Dynamic Binary Instrumentation

Dynamic Binary Instrumentation (DBI) is a method of analyzing the behavior of a binary application at runtime through the injection of instrumentation code. This instrumentation code executes as part of the normal instruction stream after being injected. In most cases, the instrumentation code will be entirely transparent to the application that it's been injected to. Analyzing an application at runtime makes it possible to gain insight into the behavior and state of an application at various points in execution. This highlights one of the key differences between static binary analysis and dynamic binary analysis. Rather than considering what *may* occur, dynamic binary analysis has the benefit of operating on what actually *does* occur. This is by no means exhaustive in terms of exercising all code paths in the application, but it makes up for this by providing detailed insight into an application's concrete execution state.

The benefits of DBI have made it possible to develop some incredibly advanced tools. Examples where DBI might be used include runtime profiling, visualization, and optimization tools. DBI implementations generally fall into two categories: light-weight or heavy-weight. A light-weight DBI operates on the architecture-specific instruction stream and state when performing analysis. A heavy-weight DBI operates on an abstract form of the instruction stream and state. An example a heavy-weight DBI is Valgrind which performs analysis on an intermediate representation of the machine state[12, 8]. An example of a light-weight DBI is DynamoRIO which performs analysis using the architecture-specific state[4]. The benefit of a heavy-weight DBI over a light-weight DBI is that analysis code written against the intermediate representation is immediately portable to other architectures, whereas light-weight DBI analysis implementations must be fine-tuned to work with individual architectures. While Valgrind is a novel and interesting implementation, it is currently not supported on Windows. For this reason, attention will be given to DynamoRIO for the remainder of this paper[1].

DynamoRIO is an example of a DBI framework that allows custom instrumentation code to be integrated in the form of dynamic libraries. The tool itself is a combination of Dynamo, a dynamic optimization engine developed by researchers at HP, and RIO, a runtime introspection and optimization engine developed by MIT. The fine-grained details of the implementation of DynamoRIO are outside of the scope of this paper, but it's important to understand the basic concepts[3].

At a high-level, *figure 1* from *Transparent Binary Optimization* provides a great visualization of the process employed by Dynamo[3]. In concrete terms, Dynamo works by processing an instruction stream as it executes. To accomplish this, Dynamo assumes responsibility for the execution of the instruction stream. It uses a disassembler to identify the point of the next branch instruction in the code that is about to be executed. The set of instructions disassembled is referred to as a fragment (although, it's more commonly known as a basic block). If the target of the branch instruction is in Dynamo's fragment cache, it executes the (potentially optimized) code in the

[1] There are many additional DBI frameworks and details, but for the sake of limiting scope these will not be discussed. The reader should consult reference material to learn more about this subject[12]

fragment cache. When this code completes, it returns control to Dynamo to disassemble the next fragment. If at some point Dynamo encounters a branch target that is not in its fragment cache, it will add it to the fragment cache and potentially optimize it. This is the perfect opportunity for instrumentation code to be injected into the optimized fragment that is generated for a branch target. Injecting instrumentation code at this level is entirely transparent to the application. While this is an oversimplification of the process used by DynamoRIO, it should at least give some insight into how it functions.

One of the best features of DynamoRIO from an analysis standpoint is that it provides a framework for inserting instrumentation code during the time that a fragment is being inserted into the fragment cache. This is especially useful for the purposes of intercepting memory accesses within an application. When a fragment is being created, DynamoRIO provides analysis libraries with the instructions that are to be included in the fragment that is generated. To optimize for performance, DynamoRIO provides multiple levels of disassembly information. At the most optimized level, only very basic information about the instructions is provided. At the least optimized level, very detailed information about the instructions and their operands can be obtained. Analysis libraries are free to control the level of information that they retrieve. Using this knowledge of DynamoRIO, it is now possible to consider how one might design an analysis library that is able to intercept memory reads and writes while an application is executing.

2.1.1 Design

DBI, and DynamoRIO in particular, make designing a solution that can intercept memory reads and writes fairly trivial. The basic design involves having an analysis library that scans the instructions within a fragment that is being created. When an instruction that accesses memory is encountered, instrumentation code can be inserted prior to the instruction. The instrumentation code can be composed of instructions that notify an instrumentation function of the memory operand that is about to be read from or written to. This has the effect of causing the instrumentation function to be called when the fragment is executed. These few steps are really all that it takes instrument the memory access behavior of an application as it executes using DynamoRIO.

2.1.2 Implementation

The implementation of the DBI approach is really just as easy as the design description makes it sound. To cooperate with DynamoRIO, an analysis library must implement a well-defined routine named `dynamorio_basic_block` which is called by DynamoRIO when a fragment is being created. This routine is passed an instruction list which contains the set of instructions taken from the native binary. Using this instruction list, the analysis library can make a determination as to whether or not any of the operands of an instruction either explicitly or implicitly reference memory. If an instruction does access memory, then instrumentation code must be inserted.

Inserting instrumentation code with DynamoRIO is a pretty painless process. DynamoRIO provides a number of macros that encapsulate the process of creating and inserting instructions into the instruction list. For example, `INSTR_CREATE_add` will create an add instruction with a specific set of arguments and `instrlist_meta_preinsert` will insert an instruction prior to another instruction within the instruction list.

A proof of concept implementation is included with the source code provided along with this paper.

2.1.3 Considerations

This approach is particularly elegant thanks to the concepts of dynamic binary instrumentation and to DynamoRIO itself for providing an elegant framework that supports inserting instrumentation code into the fragment cache. Since DynamoRIO is explicitly designed to be a runtime optimization engine, the

fact that the instrumentation code is cached within the fragment cache means that it gains the benefits of DynamoRIO's fragment optimization algorithms. When compared to alternative approaches, this approach also has significantly less overhead once the fragment cache begins to become populated. This is because all of the instrumentation code is placed entirely inline with the application code that is executing rather than having to rely on alternative means of interrupting the normal course of program execution. Still, this approach is not without its set of considerations. Some of these considerations are described below:

1. *Requires the use of a disassembler*
   DynamoRIO depends on its own internal disassembler. This can be a source of problems and limitations.

2. *Self-modifying and dynamic code*
   Self-modifying and dynamically generated code can potentially cause problems with DynamoRIO.

3. *DynamoRIO is closed source*
   While this has nothing to do with the actual concept, the fact that DynamoRIO is closed source can be limiting in the event that there are issues with DynamoRIO itself.

2.2 Page Access Interception

The hardware paging features of the x86 and x64 architectures represent a potentially useful means of obtaining information about the memory access behavior of an application. This is especially true due to the well-defined actions that the processor takes when a reference is made to a linear address whose physical page is either not present or has had its access restricted. In these cases, the processor will assert the page fault interrupt (`0x0E`) and thereby force the operating system to attempt to gracefully handle the virtual memory reference. In Windows, the page fault interrupt is handled by `nt!KiTrap0E`. In most cases, `nt!KiTrap0E` will issue a call into `nt!MmAccessFault` which is responsible for making a determination about the nature of the memory reference that occurred. If the memory reference fault was a result of an access restriction, `nt!MmAccessFault` will return an access violation error code (`0xC0000005`). When an access violation occurs, an exception record is generated by the kernel and is then passed to either the user-mode exception dispatcher or the kernel-mode exception dispatcher depending on which mode the memory access occurred in. The job of the exception dispatcher is to give a thread an opportunity to gracefully recover from the exception. This is accomplished by providing each of the registered or vectored exception handlers with the exception information that was collected when the page fault occurred. If an exception handler is able to recover, execution of the thread can simply restart where it left off. Using the principles outlined above, it is possible to design a system that is capable of both trapping and handling memory references to specific pages in memory during the course of normal process execution.

2.2.1 Design

The first step that must be taken to implement this system involves identifying a method that can be used to trap references to arbitrary pages in memory. Fortunately, previous work has done much to identify some of the different approaches that can be taken to accomplish this[9, 5]. For the purposes of this paper, one of the most useful approaches centers around the ability to define whether or not a page is restricted from user-mode access. This is controlled by the `Owner` bit in a linear address' *page table entry* (PTE)[6]. When the `Owner` bit is set to 0, the page can only be accessed at privilege level 0. This effectively restricts access to kernel-mode in all modern operating systems. Likewise, when the `Owner` bit is set to 1, the page can be accessed from all privilege levels. By toggling the `Owner` bit to 0 in the PTEs associated with a given set of linear addresses, it is possible to trap all user-mode references to those addresses at runtime. This effectively solves the first

hurdle in implementing a solution to intercept memory access behavior.

Using the approach outlined above, any reference that is made from user-mode to a linear address whose PTE has had the `Owner` bit set to 0 will result in an access violation exception being passed to the user-mode exception dispatcher. This exception must be handled by a custom exception handler that is able to distinguish transient access violations from ones that occurred as a result of the `Owner` bit having been modified. This custom exception handler must also be able to recover from the exception in a manner that allows execution to resume seamlessly. Distinguishing exceptions is easy if one assumes that the custom exception handler has knowledge in advance of the address regions that have had their `Owner` bit modified. Given this assumption, the act of distinguishing exceptions is as simple as seeing if the fault address is within an address region that is currently being monitored. While distinguishing exceptions may be easy, being able to gracefully recovery is an entirely different matter.

To recover and resume execution with no noticeable impact to an application means that the exception handler must have a mechanism that allows the application to access the data stored in the pages whose virtual mappings have had their access restricted to kernel-mode. This, of course, would imply that the application must have some way, either direct or indirect, to access the contents of the physical pages associated with the virtual mappings that have had their PTEs modified. The most obvious approach would be to simply toggle the `Owner` bit to permit user-mode access. This has many different problems, not the least of which being that doing so would be expensive and would not behave properly in multi-threaded environments (memory accesses could be missed or worse). An alternative to updating the `Owner` bit would be to have a device driver designed to provide support to processes that would allow them to read the contents of a virtual address at privilege level 0. However, having the ability to read and write memory through a driver means nothing if the results of the operation cannot be factored back into the instruction that triggered the exception.

Rather than attempting to emulate the read and write access, a better approach can be used. This approach involves creating a second virtual mapping to the same set of physical pages described by the linear addresses whose PTEs were modified. This second virtual mapping would behave like a typical user-mode memory mapping. In this way, the process' virtual address space would contain two virtual mappings to the same set of physical pages. One mapping, which will be referred to as the *original* mapping, would represent the user-mode inaccessible set of virtual addresses. The second mapping, which will be referred to as the *mirrored* mapping, would be the user-mode accessible set of virtual addresses. By mapping the same set of physical pages at two locations, it is possible to transparently redirect address references at the time that exceptions occur. An important thing to note is that in order to provide support for mirroring, a disassembler must be used to figure out which registers need to be modified.

To better understand how this could work, consider a scenario where an application contains a `mov [eax], 0x1` instruction. For the purposes of this example, assume that the `eax` register contains an address that is within the *original* mapping as described above. When this instruction executes, it will lead to an access violation exception being generated as a result of the PTE modifications that were made to the original mapping. When the exception handler inspects this exception, it can determine that the fault address was one that is contained within the original mapping. To allow execution to resume, the exception handler must update the `eax` register to point to the equivalent address within the *mirrored* region. Once it has altered the value of `eax`, the exception handler can tell the exception dispatcher to continue execution with the now-modified register information. From the perspective of an executing application, this entire operation will occur transparently. Unfortunately, there's still more work that needs to be done in order to ensure that the application continues to execute properly after the exception dispatcher con-

tinues execution.

The biggest problem with modifying the value of a register to point to the mirrored address is that it can unintentionally alter the behavior of subsequent instructions. For example, the application may not function properly if it assumes that it can access other non-mirrored memory addresses relative to the address stored within `eax`. Not only that, but allowing `eax` to continue to be accessed through the mirrored address will mean that subsequent reads and writes to memory made using the `eax` register will be missed for the time that `eax` contains the mirrored address.

In order to solve this problem, it is necessary to come up with a method of restoring registers to their original value after the instruction executes. Fortunately, the underlying architecture has built-in support that allows a program to be notified after it has executed an instruction. This support is known as single-stepping. To make use of single-stepping, the exception handler can set the trap flag (`0x100`) in the saved value of the `eflags` register. When execution resumes, the processor will generate a single step exception after the original instruction executes. This will result in the custom exception handler being called. When this occurs, the custom exception handler can determine if the single step exception occurred as a result of a previous mirroring operation. If it was the result of a mirroring operation, the exception handler can take steps to restore the appropriate register to its original value.

Using these four primary steps, a complete solution to the problem of intercepting memory accesses can be formed. First, the `Owner` bit of the PTEs associated with a region of virtual memory can be set to 0. This will cause user-mode references to this region to generate an access violation exception. Second, an additional mapping to the set of physical pages described the original mapping can be created which is accessible from user-mode. Third, any access violation exceptions that reach the custom exception handler can be inspected. If they are the result of a reference to a region that is being tracked, the register contents of the thread context can be adjusted to reference the user-accessible mirrored mapping. The thread can then be single-stepped so that the fourth and final step can be taken. When a single-step exception is generated, the custom exception handler can restore the original value of the register that was modified. When this is complete, the thread can be allowed to continue as if nothing had happened.

2.2.2 Implementation

An implementation of this approach is included with the source code released along with this paper. This implementation has two main components: a kernel-mode driver and a user-mode DLL. The kernel-mode driver provides a device object interface that allows a user-mode process to create a mirrored mapping of a set of physical pages and to toggle the `Owner` bit of PTEs associated with address regions. The user-mode DLL is responsible for implementing a vectored exception handler that takes care of processing access violation exceptions by mirroring the address references to the appropriate mirrored region. The user-mode DLL also exposes an API that allows applications to create a *memory mirror*. This abstracts the entire process and makes it simple to begin tracking a specific memory region. The API also allows applications to register callbacks that are notified when an address reference occurs. This allows further analysis of the memory access behavior of the application.

2.2.3 Considerations

While this approach is most definitely functional, it comes with a number of caveats that make it sub-optimal for any sort of large-scale deployment. The following considerations are by no means all-encompassing, but some of the more important ones have been enumerated below:

1. *Unsafe modification of PTEs*
   It is not safe to modify PTEs without acquiring certain locks. Unfortunately, these locks are not exported and are therefore inaccessible to third party drivers.

2. *Large amount of overhead*
   The overhead associated with having to take a page fault and pass the exception on to the be handled by user-mode is substantial. Memory access time with respect to the application could jump from nanoseconds to micro or even milli seconds.

3. *Requires the use of a disassembler*
   Since this approach relies on mirroring memory references from one virtual address to another, a disassembler has to be used to figure out which registers need to be modified with the mirrored address. Any time a disassembler is needed is an indication that things are getting fairly complicated.

4. *Cannot track memory references to all addresses*
   The fact that this approach relies on locking physical pages prevents it from feasibly tracking all memory references. In addition, because the thread stack is required to be valid in order to dispatch exceptions, it's not possible to track reads and writes to thread stacks using this approach.

2.3 Null Segment Interception

Segmentation is an extremely old feature of the x86 architecture. Its purpose has been to provide software with the ability to partition the address space into distinct segments that can be referenced through a 16-bit *segment selector*. Segment selectors are used to index either the *Global Descriptor Table* (GDT) or the *Local Descriptor Table* (LDT). Segment descriptors convey information about all or a portion of the address space. On modern 32-bit operating systems, segmentation is used to set up a flat memory model (primarily only used because there is no way to disable it). This is further illustrated by the fact that the x64 architecture has effectively done away with the `ES`, `DS`, and `SS` segment registers in 64-bit mode[1]. While segment selectors are primarily intended to make it possible to access memory, they can also be used to prevent access to it.

2.3.1 Design

Segmentation is one of the easiest ways to trap memory accesses. The majority of instructions which reference memory implicitly use either the `DS` or `ES` segment registers to do so. The one exception to this rule are instructions that deal with the stack. These instructions implicitly use the `SS` segment register. There are a few different ways one can go about causing a general protection fault when accessing an address relative to a segment selector, but one of the easiest is to take advantage of the *null selector*. The null selector, `0x0`, is a special segment selector that will always cause a general protection fault when using it to reference memory. By loading the null selector into `DS`, for example, the `mov [eax], 0x1` instruction would cause a general protection fault when executed. Using the null selector solves the problem of being able to intercept memory accesses, but there still needs to be some mechanism to allow the application to execute normally after intercepting the memory access.

When a general protection fault occurs in user-mode, the kernel generates an access violation exception and passes it off to the user-mode exception dispatcher in much the same way as was described in 2.2. Registering a custom exception handler makes it possible to catch this exception and handle it gracefully. To handle this exception, the custom exception handler must restore `DS` and `ES` segment registers to valid segment selectors by updating the thread context record associated with the exception. On 32-bit versions of Windows, the segment registers should be restored to `0x23`. Once the the segment registers have been updated, the exception dispatcher can be told to continue execution. However, before this happens, there is an additional step that must be taken.

It is not enough to simply restore the segment registers and then continue execution. This would lead to subsequent reads and writes being missed as a result of the `DS` and `ES` segment registers no longer pointing to the null selector. To address this, the custom exception handler should toggle the trap flag in the context record prior to continuing execution. Setting

the trap flag will cause the processor to generate a single step exception after the instruction that generated the general protection fault executes. This single step exception can then be processed by the custom exception handler to reset the DS and ES segment registers to the null selector. After the segment registers have been updated, the trap flag can be disabled and execution can be allowed to continue. By following these steps, the application is able to make forward progress while also making it possible to trap all memory reads and writes that use the DS and ES segment registers.

2.3.2 Implementation

The implementation for this approach involves registering a vectored exception handler that is able to handle the access violation and single step exceptions that are generated. Since this approach relies on setting the segment registers DS and ES to the null selector, an implementation must take steps to update the segment register state for each running thread in a process and for all new threads as they are created. Updating the segment register state for running threads involves enumerating running threads in the calling process using the toolhelp library. For each thread that is not the calling thread, the SetThreadContext routine can be used to update segment registers. The calling thread can update the segment registers using native instructions. To alter the segment registers for new threads, the DLL_THREAD_ATTACH notification can be used. Once all threads have had their DS and ES segment registers updated, memory references will immediately begin causing access violation exceptions.

When these access violation exceptions are passed to the vectored exception handler, appropriate steps must be taken to restore the DS and ES segment registers to a valid segment selector, such as 0x23. This is accomplished by updating the SegDs and SegEs segment registers in the CONTEXT structure that is passed in association with an exception. In addition to updating these segment registers, the trap flag (0x100) must also be set in the EFlags register so that the DS and ES segment registers can be restored to the null selector in order to trap subsequent memory accesses. Setting the trap flag will lead to a single step exception after the instruction that generated the access violation executes. When the single step exception is received, the SegDs and SegEs segment registers can be restored to the null selector.

These few steps capture the majority of the implementation, but there is a specific Windows nuance that must be handled in order for this to work right. When the Windows kernel returns to a user-mode process after a system call has completed, it restores the DS and ES segment selectors to their normal value of 0x23. The problem with this is that without some way to reset the segment registers to the null selector after a system call returns, there is no way to continue to track memory accesses after a system call. Fortunately, there is a relatively painless way to reset the segment registers after a system call returns. On Windows XP SP2 and more recent versions of Windows, the kernel determines where to transfer control to after a system call returns by looking at the function pointer stored in the shared user data memory mapping. Specifically, the SystemCallReturn attribute at 0x7ffe0304 holds a pointer to a location in ntdll that typically contains just a ret instruction as shown below:

```
0:001> u poi(0x7ffe0304)
ntdll!KiFastSystemCallRet:
7c90eb94 c3              ret
7c90eb95 8da42400000000  lea     esp,[esp]
7c90eb9c 8d642400        lea     esp,[esp]
```

Replacing this single ret instruction with code that resets the DS and ES registers to the null selector followed by a ret instruction is enough to make it possible to continue to trap memory accesses after a system call returns. However, this replacement code should not take these steps if a system call occurs in the context of the exception dispatcher, as this could lead to a nesting issue if anything in the exception dispatcher references memory, which is very likely.

An implementation of this approach is included with the source code provided along with this paper.

2.3.3 Considerations

There are a few considerations that should be noted about this approach. On the positive side, this approach is unique when compared to the others described in this paper due to the fact that, in principle, it should be possible to use it to trap memory accesses in kernel-mode, although it is expected that the implementation may be much more complicated. This approach is also much simpler than the other approaches in that it requires far less code. While these are all good things, there are some negative considerations that should also be pointed out. These are enumerated below:

1. *Will not work on x64*
   The segmentation approach described in this section will not work on x64 due to the fact that the `DS`, `ES`, and even `SS` segment selectors are effectively ignored when the processor is in 64-bit mode[1].

2. *Significant performance overhead*
   Like many of the other approaches, this one also suffers from significant performance overhead involved in having to take a general protection and debug exception fault for every address reference. This approach could be be further optimized by creating a custom LDT entry (using `NtSetLdtEntries`) that describes a region whose base address is 0 and length is n where `n` is just below the address of the region(s) that should be monitored. This would have the effect of allowing memory accesses to succeed within the lower portion of the address space and fail in the higher portion (which is being monitored). It's important to note that the base address of the LDT entry must be zero. This is problematic since most of the regions that one would like to monitor (heap) are allocated low in the address space. It would be possible to work around this issue by having `NtAllocateVirtualMemory` allocate using `MEM_TOP_DOWN`.

3. *Requires a disassembler*
   Unfortunately, this approach also requires the use of a disassembler in order to extract the effective address that caused the access violation exception to occur. This is necessary because general protection faults that occur due to a segment selector issue generate exception records that flag the fault address as being `0xffffffff`. This makes sense in the context that without a valid segment selector, there is no way to accurately calculate the effective address. The use of a disassembler means that the code is inherently more complicated than it would otherwise need to be. There may be some way to craft a special LDT entry that would still make it possible to determine the address that cause the fault, but the author has not investigated this.

3 Potential Uses

The ability to intercept an application's memory accesses is an interesting concept but without much use beyond simple statistical and visual analysis. Even though this is the case, the data that can be collected by analyzing memory access behavior can make it possible to perform much more extensive forms of dynamic binary analysis. This section will give a brief introduction to some of the hypothetical areas that might benefit from being able to understand the memory access behavior of an application.

3.1 Data Propagation

Being able to gain knowledge about the way that data propagates throughout an application can provide extremely useful insights. For example, understanding data propagation can give security researchers an idea of the areas of code that are affected, either directly or indirectly, by a buffer that is received from a network socket. In this context, having knowledge about areas affected by data would be much more valuable than simply understanding the code paths that are taken as a result of the buffer being received. Though the two may seem closely related, the areas of code affected by a buffer that is received should actually

be restricted to a subset of the overall code paths taken.

Even if understanding data propagation within an application is beneficial, it may not be clear exactly how analyzing memory access behavior could make this possible. To understand how this might work, it's best to think of memory access in terms of its two basic operations: read and write. In the course of normal execution, any instruction that reads from a location in memory can be said to be dependent on the last instruction that wrote to that location. When an instruction writes to a location in memory, it can be said that any instructions that originally wrote to that location no longer have claim over it. Using these simple concepts, it is possible to build a dependency graph that shows how areas of code become dependent on one another in terms of a reader/writer relationship. This dependency graph would be dynamic and would change as a program executes just the same as the data propagation within an application would change.

At this point in time, the author has developed a very simple implementation based on the DBI strategy outlined in this paper. The current implementation is in need of further refinement, but it is capable of showing reader/writer relationships as the program executes. This area is ripe for future research.

3.2 Memory Access Isolation

From a visualization standpoint, it might be interesting to be able to show with what degrees of code-level isolation different regions of memory are accessed. For example, being able to show what areas of code touch individual heap allocations could provide interesting insight into the containment model of an application that is being analyzed. This type of analysis might be able to show how well designed the application is by inferring code quality based on the average number of areas of code that make direct reference to unique heap allocations. Since this concept is a bit abstract, it might make sense to discuss a more concrete example.

One example might involve an object-oriented C++ application that contains multiple classes such as Circle, Shape, Triangle, and so on. In the first design, the application allows classes to directly access the attributes of instances. In the second design, the application forces classes to reference attributes through public getters and setters. Using memory access behavior to identify code-level isolation, the first design might be seen as a poor design due to the fact that there will be many code locations where unique heap allocations (class instances) have the contents of their memory accessed directly. The second design, on the other hand, might be seen as a more robust design due to the fact that the unique heap allocations would be accessed by fewer places (the getters and setters).

It may actually be the case that there's no way to draw a meaningful conclusion by analyzing code-level isolation of memory accesses. One specific case that was raised to the author involved how the use of inlining or aggressive compiler optimizations might incorrectly indicate a poor design. Even though this is likely true, there may be some knowledge that can be obtained by researching this further. The author is not presently aware of an implementation of this concept but would love to be made aware if one exists.

3.3 Thread Data Consistency

Programmers familiar with the pains of thread deadlocks and thread-related memory corruption should be well aware of how tedious these problems can be to debug. By analyzing memory access behavior in conjunction with some additional variables, it may be possible to make determinations as to whether or not a memory operation is being made in a thread safe manner. At this point, the author has not defined a formal approach that could be taken to achieve this, but a few rough ideas have been identified.

The basic idea behind this approach would be to combine memory access behavior with information about the thread that the access occurred in and the set of locks that were acquired when the memory access oc-

curred. Determining which locks are held can be as simple as inserting instrumentation code into the routines that are used to acquire and release locks at runtime. When a lock is acquired, it can be pushed onto a thread-specific stack. When the lock is released, it can be removed. The nice thing about representing locks as a stack is that in almost every situation, locks should be acquired and released in symmetric order. Acquiring and releasing locks asymmetrically can quickly lead to deadlocks and therefore can be flagged as problematic.

Determining data consistency is quite a bit trickier, however. An analysis library would need some means of historically tracking read and write access to different locations in memory. Still, determining what might be a data consistency issue from this historical data is challenging. One example of a potential data consistency issue might be if two writes occur to a location in memory from separate threads without a common lock being acquired between the two threads. This isn't guaranteed to be problematic, but it is at the very least be indicative of a potential problem. Indeed, it's likely that many other types of data consistency examples exist that may be possible to capture in relation to memory access, thread context, and lock ownership.

Even if this concept can be made to work, the very fact that it would be a runtime solution isn't a great thing. It may be the case that code paths that lead to thread deadlocks or thread-related corruption are only executed rarely and are hard to coax out. Regardless, the author feels like this represents an interesting area of future research.

4 Previous Work

The ideas described in this paper benefit greatly from the concepts demonstrated in previous works. The memory mirroring concept described in 2.2 draws heavily from the PaX team's work relating to their VMA mirroring and software-based non-executable page implementations[9]. Oded Horovitz provided an implementation of the paging approach for Windows and applied it to application security[5]. In addition, there have been other examples that use concepts similar to those described by PaX to achieve additional results, such as OllyBone, ShadowWalker, and others[11, 10]. The use of DBI in 2.1 for memory analysis is facilitated by the excellent work that has gone into DynamoRIO, Valgrind, and indeed all other DBI frameworks[4, 12].

It should be noted that if one is strictly interested in monitoring writes to a memory region, Windows provides a built-in feature known as a write watch. When allocating a region with `VirtualAlloc`, the `MEM_WRITE_WATCH` flag can be set. This flag tells the kernel to track writes that occur to the region. These writes can be queried at a later point in time using `GetWriteWatch`[7].

It is also possible to use guard pages and other forms of page protection, such as `PAGE_NOACCESS`, to intercept memory access to a page in user-mode. Pedram Amini's PyDbg supports the concept of memory breakpoints which are implemented using guard pages[2]. This type of approach has two limitations that are worth noting. The first limitation involves an inability to pass addresses to kernel-mode that have had a memory breakpoint set on them (either guard page or `PAGE_NOACCESS`). If this occurs it can lead to unexpected behavior, such as by causing a system call to fail when referencing the user-mode address. This would not trigger an exception in user-mode. Instead, the system call would simply return `STATUS_ACCESS_VIOLATION`. As a result, an application might crash or otherwise behave improperly. The second limitation is that there may be consequences in multi-threaded environments where memory accesses are missed.

5 Conclusion

The ability to analyze the memory access behavior of an application at runtime can provide additional insight into how an application works. This insight

might include learning more about how data propagates, deducing the code-level isolation of memory references, identifying potential thread safety issues, and so on. This paper has described three strategies that can be used to intercept memory accesses within an application at runtime.

The first approach relies on Dynamic Binary Instruction (DBI) to inject instrumentation code before instructions that access memory locations. This instrumentation code is then capable of obtaining information about the address being referenced when instructions are executed.

The second approach relies on hardware paging features supported by the x86 and x64 architecture to intercept memory accesses. This works by restricting access to a virtual address range to kernel-mode access. When an application attempts to reference a virtual address that has been marked as such, an exception is generated that is then passed to the user-mode exception dispatcher. A custom exception handler can then inspect the exception and take the steps necessary to allow execution to continue gracefully after having tracked the memory access.

The third approach uses the segmentation feature of the x86 architecture to intercept memory accesses. It does this by loading the `DS` and `ES` segment registers with the null selector. This has the effect of causing instructions which implicitly use these registers to generate a general protection fault when referencing memory. This fault results in an access violation exception being generated that can be handled in much the same way as the hardware paging approach.

It is hoped that these strategies might be useful to future research which could benefit from collecting memory access information.

References

[1] AMD. *AMD64 Architecture Programmer's Manual: Volume 2 System Programming.* http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf; accessed 5/2/2007.

[2] Amini, Pedram. *PaiMei.* http://pedram.redhive.com/PaiMei/docs/; accessed 5/10/2007.

[3] Bala, Duesterwald, Banerija. *Transparent Dynamic Optimization.* http://www.hpl.hp.com/techreports/1999/HPL-1999-77.pdf; accessed 5/2/2007.

[4] Hewlett-Packard, MIT. *DynamoRIO.* http://www.cag.lcs.mit.edu/dynamorio/; accessed 4/30/2007.

[5] Horovitz, Oded. *Memory Access Detection.* http://cansecwest.com/core03/mad.zip; accessed 5/7/2007.

[6] Intel. *Intel Architecture Software Developer's Manual Volume 3: System Programming.* http://download.intel.com/design/PentiumII/manuals/24319202.pdf; accessed 5/1/2007.

[7] Microsoft Corporation. *GetWriteWatch.* http://msdn2.microsoft.com/en-us/library/aa366573.aspx; accessed 5/5/2007.

[8] Nethercote, Nicholas. *Dynamic Binary Analysis and Instrumentation.* http://valgrind.org/docs/phd2004.pdf; accessed 5/2/2007.

[9] PaX Team. *PAGEEXEC.* http://pax.grsecurity.net/docs/pageexec.txt; accessed 5/1/2007.

[10] Sparks, Butler. *Shadow Walker: Raising the Bar for Rootkit Detection.* https://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-sparks-butler.pdf; accessed 5/3/2007.

[11] Stewart, Joe. *Ollybone.* http://www.joestewart.org/ollybone/; accessed 5/3/2007.

[12] Valgrind. *Valgrind*.
http://valgrind.org/; accessed 4/30/2007.