Improving Software Security Analysis using Exploitation Properties

12/2007

skape
mmiller@hick.org

Abstract

Reliable exploitation of software vulnerabilities has continued to become more difficult as formidable mitigations have been established and are now included by default with most modern operating systems. Future exploitation of software vulnerabilities will rely on either discovering ways to circumvent these mitigations or uncovering flaws that are not adequately protected. Since the majority of the mitigations that exist today lack universal bypass techniques, it has become more fruitful to take the latter approach. It is in this vein that this paper introduces the concept of *exploitation properties* and describes how they can be used to better understand the exploitability of a system irrespective of a particular vulnerability. Perceived exploitability is of utmost importance to both an attacker and to a defender given the presence of modern mitigations. The ANI vulnerability (MS07-017) is used to help illustrate these points by acting as a simple example of a vulnerability that may have been more easily identified as code that should have received additional scrutiny by taking exploitation properties into consideration.

1 Introduction

Modern exploit mitigations have become formidable opponents with respect to the effect they have on reliable exploitation. Some of the more substantial modern mitigations include GuardStack (GS), SafeSEH, DEP (NX), ASLR, pointer encoding, and various heap improvements[8, 9, 10, 15, 24, 3, 4]. The fact that there have been very few public exploits that have been able to universally bypass all of these mitigations at once is a testament to the resilience of these techniques working in concert with one another. It is obvious that the absence of a given mitigation directly contributes to the exploitability of the associated code. Likewise, it is also well known that most mitigations have situations in which they will offer little to no protection[5, 16, 18, 20, 2, 4]. For instance, in certain cases, it may be possible to perform a partial overwrite on Windows Vista to defeat ASLR due to the fact that only 15 bits of most 32-bit addresses may be affected by randomization[2, 17]. Other mitigations also have situations where they may not provide adequate coverage.

Given the fact that the majority of mitigations have known limitations, it makes sense to consider where this information might be useful. In the field of program analysis, whether it be manual, static, or dynamic, the question of scoping is often pertinent. This question typically revolves around figuring out what areas of code should be reviewed and what precedence, if any, should be assigned to different regions. Typical approaches taken to accomplish this often involve identifying code that straddles a trust boundary or performs complex operations reachable from a trust boundary. However, depending on one's perspective, this type of approach is insufficient in the face of modern mitigations because it may result in areas of code being reviewed that are adequately protected by all mitigations.

To help address this perceived deficiency, this paper introduces the concept of *exploitation properties* and describes how they can be used to provide a better understanding of the exploitability of a system if a vulnerability is found to be present. Regions of code that are found to have a number of distinct exploitation properties may be more interesting from an exploitation standpoint and therefore may warrant additional scrutiny from a program analysis perspective. The use of exploitation properties may benefit both an attacker and a defender. For example, companies may wish to perform targeted reviews on areas of code that may be more trivially exploited in an effort to prevent reliable exploits from being released in the future. Likewise, an attacker searching for a vulnerability may wish to avoid auditing regions of code that are likely to be more difficult to exploit.

Exploitation properties represent additional criteria that can be used when attempting to better understand the security aspects of a program. Annotating regions of code with exploitation properties makes it possible to use set unions and intersections to identify the subset of interesting regions of code for a partic-

ular analysis problem. For example, an attacker may wish to determine the regions of code that may permit the use of traditional stack-based buffer overflow techniques as well as permitting a partial overwrite of a return address in order to defeat ASLR. Using these two exploitation properties as criteria, a *narrowed* subset can be produced which contains only those regions which meet both criteria by intersecting those regions that have both exploitation properties. For the purpose of this paper, the term narrowing is not used in the strict mathematical sense; rather, this paper uses narrowing to describe the process of constraining the scope of analysis through the use of specific criteria.

The concept of using automated analysis as a precursor to more strenuous program analysis is certainly not new. There have been many tools ranging from the simple detection of calls to `strcpy` to much more sophisticated forms of static analysis. Still, the use of exploitation properties can be seen as an additional set of data points which may be useful in the context of program analysis given the hypothesis that most reliably exploitable security vulnerabilities are being pushed into areas of code that are less affected by mitigations.

The concept of exploitation properties is presented as follows. §2 categorizes and defines a limited number of concrete exploitation properties. §3 provides a concrete example of using exploitation properties to help identify the function that contained the ANI vulnerability. §4 describes some potential ways in which exploitation properties can be applied. §5 gives a brief description of future work involving exploitation properties.

2 Exploitation Properties

Exploitation properties describe the ease with which an arbitrary vulnerability might be exploited. An understanding of a system's perceived exploitability can provide useful insights when attempting to establish the risk factors associated with it[1]. It is important to note that exploitation properties do not provide any indication that a vulnerability exists; instead, they are only meant to convey information about how easily a vulnerability *could* be exploited. The concept of an exploitation property can be broken into different categories which are tied to the configuration or context that the property is associated with. Examples of these categories include platforms, processes, binary modules, functions, and so on.

The following subsections provide concrete examples to better illustrate the concept of an exploitation property. These examples are given by showing what implications a property has with respect to exploitation as well as how a property might be derived. It should be noted that the examples given in this paper do not represent a complete, exhaustive set of exploitation properties.

2.1 Platform Properties

Exploitation properties associated with a platform are meant to illustrate how easily a vulnerability may be exploited when a given platform configuration, such as the operating system or architecture, is used. For example, Windows 2000 does not include support for enforcing non-executable pages. This implies that any vulnerability found within an application that runs in the context of the Windows 2000 platform may be exploited more easily. An understanding of exploitation properties that are associated with a platform may be useful when attempting to assess the risk of applications that might run on multiple platforms. There are many other examples of exploitation properties that are tied to platforms. In order to limit the scope of this document, platform exploitation properties are not discussed at length.

[1] An example of this can be seen in threat modeling where the DREAD model of classifying risk includes a high-level evaluation of exploitability as one of the risk factors[14]

2.2 Process Properties

Process exploitation properties carry some information about how easily vulnerabilities found within the context of a running process may be exploited. For example, Internet Explorer running on 32-bit versions of Windows Vista do not make use of hardware-enforced DEP (NX) by default. This means that any vulnerabilities found within code that runs in the context of Internet Explorer will not be protected by non-executable regions. An understanding of exploitation properties that are associated with a process context can help to provide a better understanding of the risks associated with code that may run in the context of a given process. In order to limit the scope of this document, process exploitation properties are not discussed at length.

2.3 Module Properties

Module exploitation properties are used to illustrate the effect that a particular binary module has on ease of exploitation. This category of properties is useful when attempting to identify binaries that may be more easily exploited if a vulnerability is found within them or in code that depends on them. This subsection describes two examples of module exploitation properties.

2.3.1 No Support for ASLR

Windows Vista was the first major release of Windows to include a built-in implementation of *Address Space Layout Randomization* (ASLR)[15, 24]. In order to head off potential application compatibility issues, Microsoft chose to make ASLR an opt-in feature by requiring binaries to be compiled with a new compiler switch (`/dynamicbase`)[21]. This compiler switch is responsible for setting a bit (`0x40`) in the `DllCharacteristics` that are defined within a binary. If this bit is set, the Windows kernel will attempt to randomize the base address of the binary when it is mapped into memory the first time. If the bit is not set, the binary will not have its base address randomized, although it could be relocated in memory if the binary's preferred region is already occupied by another allocation. As such, any binary that does not support ASLR may be mapped at a predictable location within a process address space at execution time. This can allow an attacker to make assumptions about the address space which may make exploitation easier if a vulnerability is found within any code that is mapped into the same address space as the module of interest.

2.3.2 No Support for SafeSEH

With Visual Studio 2003, Microsoft introduced a compile-time change known as SafeSEH which attempts to act as a mitigation for the SEH overwrite attack vector[5, 9]. SafeSEH works by adding a static list of known good exception handlers that are considered valid as metadata within a given binary. Binaries that support SafeSEH allow the exception dispatcher to perform additional checks when dispatching exceptions. The most important check involves determining if an exception handler that is found to exist within the mapped region of a given binary is actually considered to be one of the safe exception handlers. If the exception handler is not a safe exception handler, the exception dispatcher can take steps to prevent it from being called. This behavior works to mitigate the potential exploitation vector.

In order to communicate this information to the exception dispatcher, modern PE files include fields in the load config data directory which hold the offset of the safe exception handler table and the number of elements found within the table. The load config data directory contains meta data that is useful to the dynamic loader such as information about safe exception handlers, the module's global security cookie address, and so on[13]. The following output from `dumpbin.exe` illustrates what this might look like:

```
310751E0 Safe Exception Handler Table
       1 Safe Exception Handler Count
```

```
Safe Exception Handler Table

  Address
  --------
  310357D1  __except_handler4
```

Unfortunately, as with ASLR, the benefits offered by SafeSEH are not complete unless every binary that is loaded into an address space has been compiled to make use of SafeSEH. If a binary has not been compiled to make use of SafeSEH, an attacker may be able to use any address found within the binary's memory mapping as an exception handler in conjunction with an SEH overwrite.

2.4 Function Properties

Function exploitation properties convey information about how a function contributes to the exploitability of an application. For example, a function might make it possible to use certain exploitation techniques that might otherwise be prevented if mitigations were present. Alternatively, a function might simply assist in the exploitation process. Function exploitation properties are especially useful because they provide more detailed information than exploitation properties that are derived from the platform, process, or module context.

2.4.1 Absence of GuardStack

The GuardStack (GS) support included with versions of the Microsoft Visual Studio compiler since 2002 offers a compile-time mitigation to traditional stack-based buffer overflows[23]. It supports this through a combination of a random canary inserted into a stack frame at runtime and an intelligent stack frame layout algorithm. The random canary is pushed onto the stack when a function is called and then popped off the stack and validated prior to function return. If the canary does not match the expected value, it is assumed that a stack-based buffer overflow occurred and that the process should be terminated.

Since the initial release of GS support a number of techniques have been described that could be used to bypass or weaken it[5, 16, 20]. While these techniques were at one time useful or have not yet been fully realized, the author assumes that most would agree that the GS implementation provided by the most recent compiler is robust (with the exception of SEH). There is currently no publicly known universal bypass technique for GS that the author is aware of. Given this assumption, functions that are protected by GS become less interesting from the standpoint of identifying stack-based buffer overflows. On the other hand, functions that are not protected by GS can instantly be qualified as interesting targets for review. This is especially true with binaries that have been compiled with GS support but contain a number of functions that the compiler has chosen not to compile with GS protections[2].

As previous research has illustrated[27], it is possible to identify functions that have not been compiled to use GS through the use of simple static analysis tools. It is also possible to further refine the approaches described in previous research if one has symbols and one assumes that the most recent compiler was used. This can be accomplished by analyzing the call graph of an executable and noting the set of functions that do not call `__security_check_cookie`. Considered another way, the same set of functions can be identified by taking the set of all functions contained within a binary less the subset that call `__security_check_cookie`. The set of functions that is identified by either approach can be annotated with an exploitation property that indicates that they may contain stack-based buffer overflows that would not be hindered by GS.

It may also be prudent to take the compiler version that was used into consideration when analyzing binaries. This is important due to the fact that older versions of the compiler used a GS implementation that could be trivially defeated in certain circumstances[16]. For example, previous versions of

[2]This choice is made by taking into account certain conditions such as the presence or absence of local variables that are declared as fixed-size arrays

GS did not layout the stack frame in a manner that would prevent an attacker from overwriting other local variables and function arguments. In scenarios where this occurred and an overwritten local variable or parameter was dereferenced (such as by invoking a function pointer), the mitigation offered by GS would be meaningless. Thus, a secondary exploitation property could involve identifying functions where attacks such as the one described above could be possible.

2.4.2 Partial Overwrite Feasibility

One of the unique consequences of implementing Address Space Layout Randomization (ASLR) on Windows is the limitation that the system allocation granularity imposes on the number of bits that can be randomized within most memory allocations. In particular, the allocation granularity used by Windows enforces strict 16-page alignment for the base addresses of most memory mappings in user-mode. This restriction means that it is only possible to introduce entropy into the low 15 bits of the high-order 16 bits of a 32-bit memory mapping[17][3]. The low-order 16 bits remain unchanged relative to the high-order bits. This caveat means that it may be possible to perform a partial overwrite of an address and thus bypass the security features offered by ASLR[2]. However, the ability to perform a partial overwrite also relies on the presence of useful code or data within a region that is relative to the address that is being overwritten.

To visualize how this type of information might be useful, consider a scenario where an attacker is performing a partial overwrite of a return address on the stack. In this situation, it is often necessary for one or more useful opcodes to be present at an address that is 16-page relative to the return address. For example, consider a scenario where the function f may have a vulnerability that would permit a partial overwrite. In this example, f is called by h and y. In order to permit the use of a partial overwrite, a useful opcode must be found within the same 16-page aligned region that either h or y reside on. If a useful opcode is present, an exploitation property can be attached to f in order to indicate that a partial overwrite may be feasible due to the presence of a useful opcode within the same 16-page aligned region as either h or y. For example, consider the following pseudo-disassembly illustrating a case where the `call f` instruction in h is on the same 16-page region as a useful opcode:

```
... useful jmp on same 16-page region 0x14c1XXXX
0x14c1fc04  jmp esp
... entry point to h()
0x14c1a910  push ebp
0x14c1a911  mov  ebp, esp
0x14c1a914  call f
... entry point to y(), not on same 16-page region
0x137f44c8  push ebp
```

While this captures the basic concept, a better approach might be to view a binary in a different way. For example, consider the following approach to drawing the same conclusion: for each code region that contains a useful opcode, identify the subset of functions that are called from call sites within the same 16-page aligned region as the useful opcode. This has the effect of annotating all of the child functions that could potentially leverage a partial overwrite of the return address with respect to a particular collection of opcodes.

One important point that must be made about this exploitation property is that is entirely dependent upon the definition of "useful code or data". Exploitation is very much an art and it goes without saying that attempting to constrain the approaches that an attacker might make use of is likely to be folly. However, defining a known-set of useful opcodes and using that set as a base with which to draw the above conclusion can be said to be better than not doing so at all.

[3]While this may sound odd at first glance, the high-order two bits are not randomized due to the divide between kernel and user-mode. This assumes that a machine is booted without /3GB.

2.4.3 Function or Parent Registers an Exception Handler

One of the unique exploitation vectors that exists in 32-bit programs that run on Windows is known as an *SEH overwrite*[5]. An SEH overwrite makes it possible to gain control of execution flow by overwriting an exception registration record on the stack. From an exploitation perspective, the act of registering an exception handler within a function opens up the possibility of making use of an SEH overwrite. Since exception handlers are chained, the act of registering an exception handler also implicates any functions that are children of a function that registers the exception handler. This makes it possible to define an exploitation property that illustrates the possibility of an SEH overwrite being abused within the scope of a specific set of functions. Detecting this property can be as simple as signaturing the compiler generated code that is used to generate and register an exception handler within a function. An example of two functions, f and g, that would meet this criteria can be seen below:

```
void f() {
   __try {
      g();
   } __except(EXCEPTION_EXECUTE_HANDLER) {
   }
}

void g() {
   ...
}
```

In addition to this information being useful from an SEH overwrite perspective, it may also benefit an attacker in situations where an exception handler simply swallows any exceptions that are dispatched without crashing the process[1]. In the example given above, any exception that occurs in the context of g will be swallowed by f without necessarily crashing the process. This behavior may allow an attacker to retry their exploitation attempt multiple times, thus enabling a bruteforce attack that would otherwise not be feasible. This can make defeating ASLR more feasible.

2.4.4 Function is an Exception Handler

The introduction of SafeSEH as a modern compile-time mitigation has caused the particulars of how exception handlers are implemented to become more interesting. This has to do with the fact that SafeSEH restricts the set of exception handlers that may be called by the exception dispatcher to those that are specified as being valid within the scope of a given binary. As discussed previously in this paper, SafeSEH prevents traditional SEH overwrites from being able to use any address as the overwritten exception handler. While this is effective in its primary intent, there is still the possibility that a valid exception handler can be abused to make exploitation more feasible[1]. This scenario is restricted to EH3 and prior exception handlers as EH4 includes a check of a cookie before dispatching exceptions. As such, it may be useful to flag the regions of code that are associated with EH3 and prior exception handlers, including language-specific exception handlers, as being potentially interesting from an exploitation perspective.

Unfortunately, as with ASLR, the benefits offered by SafeSEH are not complete unless every binary that is loaded into a process address space has been compiled to make use of SafeSEH. If a binary has not been compiled to make use of SafeSEH, an attacker may be able to use any address found within the binary's memory mapping as an exception handler in the context of an SEH overwrite. This may make exploitation more feasible.

3 Case Study: MS07-017

The animated cursor (ANI) vulnerability was discovered by Alexander Sotirov in late 2006 and patched by Microsoft with the MS07-017 critical update in April, 2007 . Apart from being a client-side vulnerability that was exposed through web-browsers and other mediums, the ANI vulnerability was one of the first notable security issues that affected Windows Vista. It was notable due to the simple fact that

even though Microsoft had touted Windows Vista as being the most secure operating system to date, the exploits that were released for the ANI vulnerability were very reliable. These exploits were able to ignore or defeat the protections offered by mitigations such as GS, DEP, and even Vista's newest mitigation: ASLR.

To better understand how this was possible it is important to dive deeper into the details of the vulnerability itself. §3.1 gives a brief description of the ANI vulnerability and some of the techniques that were used to successfully exploit it. Following this description, §3.2 illustrates how exploitation properties, in combination with another class of properties, can be used to detect functions that may contain vulnerabilities similar to the ANI vulnerability. This is meant to help illustrate the perceived benefits of applying the concept of exploitation properties to aide in the process of identifying regions of code that may deserve additional scrutiny based on their perceived exploitability.

3.1 Background

While the ANI vulnerability was certainly unique, it was not the first time the animated cursor code was found to have a security issue. Microsoft patched an issue that was almost exactly the same as MS07-017 with MS05-002 roughly two years prior[7]. In both cases, the underlying security issue was related to a failure to properly validate input that was derived from the contents of an animated cursor file. Alexander Sotirov provided much of the initial research on the ANI vulnerability and also gave an excellent write-up to its effect[22]. This paper will only attempt to highlight the flaw.

The vulnerability itself was found in `user32!LoadAniIcon` which is responsible for processing a number of different chunks that may be contained within an animated cursor file. Each chunk is a TLV (Type-Length-Value) as described by the following structure[4]:

[4]Copied from Sotirov's write-up with permission

```
struct ANIChunk
{
    char  tag[4];        // ASCII tag
    DWORD size;          // length of data in bytes
    char  data[size];    // variable sized data
}
```

Keeping this structure in mind, the flaw itself can be seen in the abbreviated pseudo-code below as modified slightly from Sotirov's original write-up:

```
01: int LoadAniIcon(struct MappedFile* file, ...) {
02:   struct ANIChunk  chunk;
03:   struct ANIHeader header;  // 36 byte structure
04:   while (1) {
05:     // read the first 8 bytes of the chunk
06:     ReadTag(file, &chunk);
07:     switch (chunk.tag) {
08:       case 'anih':
09:         // read chunk.size bytes into header
10:         ReadChunk(file, &chunk, &header);
```

On line 6, the chunk header is read into the local variable `chunk` using `ReadTag` which populates the chunk's `tag` and `size` fields. If the chunk's tag is equal to `'anih'`, the data associated with the chunk is read into the `header` local variable using `ReadChunk` on line 10. The problem is that `ReadChunk` uses the `size` field of the chunk as the amount of data to read from the file. Since `header` is a fixed-size (36 byte) data structure and the chunk's size can be variable, a trivial stack-based buffer overflow may occur if more than 36 bytes are specified as the chunk size. In terms of the vulnerability, that's all there is to it, but the implications from an exploitation perspective are where things start to get interesting.

When attempting to exploit this vulnerability it may at first appear that all attempts to do so would be futile. Given Vista's security push, an attacker would be justified in thinking that surely the `LoadAniIcon` function is protected by a GS cookie. This point is especially justified considering the majority of all binaries shipped with Windows Vista have been compiled with GS enabled[27]. However, there are indeed circumstances where the compiler will choose to not

enable GS for a specific function. As chance would have it, the compiler chose not to enable GS for the `LoadAniIcon` function because of the simple fact that it does not contain any characteristics that would suggest that a stack-based buffer overflow might be possible (such as the use of stack-allocated arrays). This means that an attacker is able to make use of exploitation techniques that are associated with traditional stack-based buffer overflows. While this drastically increases the chances of being able to produce a reliable exploit, there are still other mitigations that are of potential concern.

Another mitigation that might be concerning in most circumstances is hardware-enforced DEP (NX). This would generally prevent an attacker from being able to run arbitrary code within regions that are not marked as executable (such as the stack and the heap). However, as fate would have it, Internet Explorer is configured to not run with DEP enabled. This immediately removes this concern from the equation for exploits that attempt to trigger the ANI vulnerability through Internet Explorer. With DEP out of the picture, ASLR becomes a weakened but still potentially significant hurdle.

While it may appear that ASLR would be challenging to defeat in most circumstances, this particular vulnerability provides an example of two different ways in which ASLR can be bypassed. The simplest approach, as taken by Sotirov, involves making use of the fact that Internet Explorer is not compiled with support for ASLR and therefore can be found at a fixed address within the address space. This allows an attacker to make use of opcodes contained within `iexplore.exe`'s memory mapping. A second approach, as taken by the author, involves using a partial overwrite to ignore the effects of ASLR completely. The details relating to how a partial overwrite works were explained in §2.4.2. In either case, an attacker is able to reliably defeat Vista's ASLR.

To compound the problem, the particulars of the context in which this vulnerability occur make it easier to exploit even without the presence of mitigations. This improved reliability comes from the fact that the `LoadAniIcon` function is wrapped in an exception handling context that simply swallows exceptions that are encountered. This makes it possible for an exploit to fail without actually crashing the process, thus allowing the attacker to try multiple times without having to worry about making a mistake that crashes the process. When all is said and done, the simplicity of the vulnerability and the ease with which mitigations could be bypassed are what lead to the ANI vulnerability being quite unique. Given the fact that this vulnerability can be so easily exploited, it is prudent to describe how it could have been detected as being a high risk function.

3.2 Detection

The ease of exploitability associated with the ANI vulnerability makes it an obvious candidate for study with respect to the exploitation properties that have been described in this paper. It should be possible to use extremely simple criteria to accomplish two things. First, the criteria must identify the `LoadAniIcon` function. Second, the criteria should be unique enough to limit the size of the narrowed subset. Reducing the subset size is beneficial as it may permit the use of more complex program analysis tools which can further constrain or explicitly identify instances of vulnerabilities. Determining the specific criteria that is needed to identify the `LoadAniIcon` function can help illustrate how one can make use of exploitation properties. Given the description of the ANI vulnerability, one can easily deduce some of the more interesting properties that it has.

An exploitation property that one might immediately observe is that the `LoadAniIcon` function does not make use of GS (§2.4.1). This makes it possible to define criteria which states that only functions that have *not* been compiled with GS should be considered. Functions that have been compiled with GS are inherently less interesting for the purpose of this exercise due to the fact that they are less likely to contain exploitable vulnerabilities.

A second property that the ANI vulnerability had with regard to exploitation was that it was possible

for an attacker to make use of a partial overwrite to defeat ASLR. The exploitation property described in §2.4.2 illustrates how one can make this determination statically. In the case of the ANI vulnerability, a partial overwrite can be performed by making use of a `jmp [ebx]` that is located within the same 16-page aligned region as the caller of `LoadAniIcon`. Thus, any functions that could potentially make use of a partial overwrite can be used as additional criteria.

At this point, a subset can be produced that is constrained to the regions of code that are annotated with the GS and partial overwrite exploitation properties. It is possible to further refine the set of functions that should ultimately be considered by studying the form that the ANI vulnerability took. The first point to note is that the stack-based buffer overflow occurred when writing beyond the bounds of a `struct` that was allocated on the stack. Furthermore, the overflow did not actually occur in the immediate context of the `LoadAniIcon` itself. Instead, the overflow was triggered by passing a pointer to the stack-allocated `struct` as a parameter when calling the function `ReadChunk`.

Based on these data points it is possible to define a third criteria. In this case, the third criteria is not an exploitation property but is instead an example of a *vulnerability property*. While not discussed in detail in this paper, many examples of vulnerability properties exist, though perhaps not categorized as such. A vulnerability property can be thought of as an annotation that illustrates whether or not a region of code has a form that is similar to that seen in vulnerabilities or has the potential of being a vulnerability. The complexity of a vulnerability property, as with the complexity of an exploitation property, can range from highly sophisticated to very simplistic.

For the purpose of this paper, a vulnerability property can be used that is very simple and imprecise but nevertheless effective at further narrowing the set of functions that should be reviewed. This property is based on whether or not a function passes a pointer to a stack-allocated variable as a parameter to a child function. This property is directly derived from the general form that the ANI vulnerability takes. At a minimum, a region of code that matches this form suggests that a vulnerability *could* be present.

Using these three properties, it should be possible to easily identify both the function that contains the ANI vulnerability as well as other functions that could contain similar vulnerabilities. However, it is important to note that this process does not produce functions that definitely have vulnerabilities. This can be plainly seen by the fact that both the vulnerable and fixed versions of the `LoadAniIcon` should be detected by the criteria described above. While this may seem to run counter to the purposes of this paper, it is important for the reader to remember that the goal of using these exploitation properties is not to identify specific instances of vulnerabilities. Instead, the goal is to identify regions of code that might warrant additional scrutiny due to the relative ease with which a vulnerability could be exploited if one is found to be present.

3.3 Test Case

The author developed an analysis tool as an extension to Microsoft's Phoenix framework in order to test the ideas described in this paper[12]. Unfortunately, the current release (July 2007 SDK) of Phoenix requires private symbol information for native binaries. This limitation prevented the author from being able to run the analysis tool across the vulnerable version of `user32.dll`. In lieu of this ability, the author chose to generate a binary containing test cases that closely mirror the form of the function containing the ANI vulnerability.

Using these test cases, the author used the features provided by the analysis tool to determine the exploitation and vulnerability properties described in the previous section and to identify the resulting subset of functions meeting all criteria. This was accomplished by first attempting to identify the subset of functions that do not contain GS within the scope of the target binary. After identifying the subset of functions without GS, a second subset was taken which consists of the functions that pass a pointer

to a stack-allocated local variable as a parameter to a child routine. This was accomplished by using Phoenix's *static single assignment* (SSA) and alias implementations to collect the requisite data flow information[12, 25]. Using this data flow information, it is possible to perform backwards data flow analysis to determine the potential storage location of the parameter being passed at each point along a given data flow path starting from the operand associated with a parameter at a call site. The analysis terminates either when a fixed point is reached or when it is determined that a pointer to a stack-allocated variable *could* be passed as the parameter.

While the previous section described the potential for using the partial overwrite exploitation property to detect the function containing the ANI vulnerability, it is not possible to create a meaningful parallel between the test binary and that of the ANI vulnerability. This is due in part to the fact that while it would certainly be possible to artificially place a useful opcode at a specific location in the test binary, it would not add any value beyond showing that it is possible to detect useful opcodes within the same 16-page aligned region as the caller of a given function. The author feels that this point is somewhat moot given the fact that it has already been proven that a partial overwrite can be used with the ANI vulnerability[6]. The only additional benefit that it could offer in this case would be to help further constrain the resultant set size. However, without being able to run this analysis against the vulnerable version of `user32.dll`, it is not possible to draw meaningful conclusions at this point in time.

3.4 Results

The results of running the analysis tool against the test binary produced the expected behavior. To illustrate this, it is helpful to consider a sampling of the functions that were analyzed. The following functions have a form that is similar to the ANI vulnerability. These functions also match the criteria described in the previous subsection. Specifically, these functions do not make use of GS and pass a pointer to a stack-allocated local variable (`var`) to a child function:

```
int tc_df_pass_local_ptr_to_callee() {
  int var;
  tc_df_pass_local_ptr_to_callee_func(&var);
  return 0;
}
int tc_df_pass_local_ptr_to_callee_alias() {
  int var;
  int *p = &var;
  tc_df_pass_local_ptr_to_callee_func(p);
  return 0;
}
int tc_df_pass_local_ptr_to_callee_alias_struct(
    struct _foo *foo) {
  int var;
  foo->ptr = &var;
  return tc_df_pass_local_ptr_to_callee_func(
    foo->ptr);
  return 0;
}
```

Additionally, a handful of different test functions were also included in the target binary in an effort to ensure that other scenarios were not improperly detected as matching the criteria. Some examples of these functions include:

```
int tc_df_pass_local_to_callee_alias() {
  int var = 2;
  int p = var;
  tc_df_pass_local_to_callee_func(p);
  return 0;
}
int tc_df_pass_local_to_callee_deref() {
  int var = 2;
  int *p = &var;
  tc_df_pass_local_to_callee_func(*p);
  return 0;
}
int tc_df_pass_heap_ptr_to_callee(struct _foo *foo) {
  tc_df_pass_local_ptr_to_callee_func(&foo->val);
  return 0;
}
```

When running the analysis tool against the target binary, the following output is shown:

```
>PhaseRunner.exe detectani.xml dfa.exe
Running phase: ANI Detection ... 1 target(s)
```

```
Displaying 3 normalizables at the
  ProgramElement.Method granularity...

00001: dfa!tc_df_pass_local_ptr_to_callee_alias
00002: dfa!tc_df_pass_local_ptr_to_callee
00003: dfa!tc_df_pass_local_ptr_to_callee_alias_struct
```

While this unfortunately does not prove that these techniques could be used to identify the function containing the ANI vulnerability, it does nevertheless hint at the potential for detecting the function containing the ANI vulnerability using its suggested exploitation and vulnerability properties. As an side, another interesting way in which this type of detection can be accomplished is through the use of Language Integrated Queries (LINQ) which are now supported in Visual Studio 2008[11]. For instance, a simple LINQ expression for the above narrowing operation can be expressed as:

```
var matches =
 from
   Method method in engine.GetScopeMethods()
 where
   !method.IsGuardStackEnabled() &&
   method.IsPassingStackLocalPtrToChild()
 select method;

foreach (var method in matches)
   Console.WriteLine("{0} matches", method);
```

4 Potential Uses

Program analysis is one area that may benefit from the use of exploitation properties. In particular, an auditor can make use of exploitation properties to assist in the process of identifying regions of code that should be audited more closely or with greater precedence. This determination can be made by using exploitation properties to understand the ease of exploitation associated with specific binaries or functions. By combining this information with other data that is collected either manually or automatically, an auditor can get a better understanding of the security aspects that are associated with a system. This is beneficial both to an attacker and a defender. An attacker can identify regions of code that would be easier to exploit and thus devote more time to auditing those regions. Likewise, a defender can use this information to the same extent but for different purposes. This type of information is especially useful to a defender who needs to balance the cost associated with performing security reviews because it should offer a better understanding of what the business cost might be if a vulnerability is found in a region of code. This cost can be derived from the negative publicity and response effort needed to cope with a flaw that is found publicly in a region of code that is widely exploited. For example, consider some of the Windows flaws that have lead to wormable issues and the cost they have had relative to other issues.

Exploitation properties may also benefit the security community by helping to identify ways in which future mitigations can be applied. This would involve analyzing regions of code that could be more easily exploited in an effort to determine what other forms of mitigations could help to protect these regions, if any. This information could be fed back to the compiler to make it possible for mitigations to be enabled that might otherwise be disabled by default. For example, a function that by default would not have GS but is subsequently found to be highly exploitable may benefit from having the compiler insert GS.

5 Future Work

While this paper has defined exploitation properties and described a handful of concrete examples, it has not attempted to formally define the correlation between exploitation properties and the exploitation techniques they are associated with. Future research will attempt to concretely define this relationship as it should lead to a better understanding of the variables that permit the use of various exploitation techniques. Using more formal definitions of exploitation properties, a larger scale case study can be completed

which collects data about the effect of using exploitation properties to improve program understanding for a variety of purposes. The author views exploitation properties as being one component in a larger model. This larger model could be used to join major areas of study within computer security including attack surface analysis, vulnerability analysis, and exploitation analysis to form a more complete understanding of the true risks associated with a system.

6 Conclusion

This paper has introduced the general concept of *exploitation properties* and described how they can be used to better understand the exploitability of a system. The purpose of an exploitation property is to help convey the ease with which a vulnerability might be exploited *if* one is found to be present. Exploitation properties can be broken down into different categories based on the configuration or context that a given property is associated from. These categories include operating platforms, running processes, binary modules, and functions.

Exploitation properties can be used to provide an alternative understanding of an application's attack surface from the perspective of which areas would be most trivially exploited. This can allow an attacker to focus on finding security issues in code that would be more easily exploited. Likewise, a defender can draw the same conclusions and direct resources of their own at reviewing the associated code. It may also be possible to use this information to augment existing mitigations or to come up with new mitigations. A contrived example based on the form of the ANI vulnerability was used to illustrate an automated approach to extracting exploitation properties and using them to help identify a constrained subset of regions of code that meet a specific criteria. Future research will attempt to better define the extent of exploitation properties and their uses.

References

[1] Dowd, M., Metha, N., McDonald, J. *Breaking C++ Applications.* https://www.blackhat.com/presentations/bh-usa-07/Dowd_McDonald_and_Mehta/Whitepaper/bh-usa-07-dowd_mcdonald_and_mehta.pdf

[2] Durden, Tyler. *Bypassing PaX ASLR Protection.* July, 2002. http://www.phrack.org/issues.html?issue=59&id=9

[3] Howard, Michael. *Protecting against Pointer Subterfuge (Kinda!).* http://blogs.msdn.com/michael_howard/archive/2006/01/30/520200.aspx

[4] Johnson, Richard. *Windows Vista: Exploitation Countermeasures.* http://rjohnson.uninformed.org/

[5] Litchfield, David. *Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server.* http://www.nextgenss.com/papers/defeating-w2k3-stack-protection.pdf

[6] Metasploit. *Exploiting the ANI vulnerability on Vista.* http://blog.metasploit.com/2007/04/exploiting-ani-vulnerability-on-vista.html

[7] Microsoft Corporation. *Microsoft Security Bulletin MS05-002.* Jan, 2005. http://www.microsoft.com/technet/security/Bulletin/MS05-002.mspx

[8] Microsoft Corporation. */GS (Buffer Security Check).* http://msdn2.microsoft.com/en-us/library/8dbf701c(VS.80).aspx

[9] Microsoft Corporation. */SAFESEH (Image has Safe Exception Handlers).* http://msdn2.microsoft.com/en-us/library/9a89h429.aspx

[10] Microsoft Corporation. *A detailed description of the Data Execution Prevention (DEP) feature.* http://support.microsoft.com/kb/875352

[11] Microsoft Corporation. *The LINQ Project*. http://msdn2.microsoft.com/en-us/netframework/aa904594.aspx

[12] Microsoft Corporation. *Phoenix*. http://research.microsoft.com/phoenix/

[13] Microsoft Corporation. *Microsoft Portable Executable and Object File Format Specification*. http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/pecoff_v8.doc

[14] Microsoft Corporation. *Threat Modeling*. June, 2003. http://msdn2.microsoft.com/en-us/library/aa302419.aspx

[15] PaX Team. *ASLR*. http://pax.grsecurity.net/docs/aslr.txt

[16] Ren, Chris et al. *Microsoft Compiler Flaw Technical Note*. http://www.cigital.com/news/index.php?pg=art&artid=70

[17] Rahbar, Ali. *An analysis of Microsoft Windows Vistas ASLR*. Oct, 2006. http://www.sysdream.com/articles/Analysis-of-Microsoft-Windows-Vista's-ASLR.pdf

[18] skape, Skywing. *Bypassing Windows Hardware-enforced DEP*. http://www.uninformed.org/?v=2&a=4&t=sumry

[19] skape. *Preventing the Exploitation of SEH Overwrites*. http://www.uninformed.org/?v=5&a=2&t=sumry

[20] skape. *Reducing the Effective Entropy of GS Cookies*. http://www.uninformed.org/?v=7&a=2&t=sumry

[21] Skywing. *Vista ASLR is not on by default for image base addresses*. http://www.nynaeve.net/?p=100

[22] Sotirov, Alexander. *Windows Animated Cursor Stack Overflow Vulnerability*. March, 2007. http://www.determina.com/security.research/vulnerabilities/ani-header.html

[23] Wikipedia. *Stack-smashing protection*. http://en.wikipedia.org/wiki/Stack-smashing_protection

[24] Wikipedia. *Address space layout randomization*. http://en.wikipedia.org/wiki/ASLR

[25] Wikipedia. *Static single assignment form*. http://en.wikipedia.org/wiki/Static_single_assignment_form

[26] University of Wisconsin. *Wisconsin Program-Slicing Project's Home Page*. http://www.cs.wisc.edu/wpis/html/

[27] Whitehouse, Ollie. *Analysis of GS protections in Microsoft Windows Vista*. http://www.symantec.com/avcenter/reference/GS_Protections_in_Vista.pdf