Exploiting the Otherwise Non-exploitable on Windows

Taking Another Stab at the Unhandled Exception Filter

May 2006

Skywing
Skywing@valhallalegends.com

skape
mmiller@hick.org

Contents

1 Foreword **2**

2 Introduction **3**

3 Understanding Unhandled Exception Filters **5**
3.1 Setting the Top-Level UEF 5
3.2 Handling Unhandled Exceptions 7
3.3 Uses for Unhandled Exception Filters 8

4 Gaining Control of the Unhandled Exception Filter **9**

5 Case Study: Internet Explorer **12**

6 Mitigation Techniques **17**
6.1 Behavioral Change to SetUnhandledExceptionFilter 17
6.2 Prevent Setting of non-image UEF 18
6.3 Prevent Execution of non-image UEF 18

7 Future Research **20**

8 Conclusion **22**

Chapter 1

Foreword

Abstract: This paper describes a technique that can be applied in certain situations to gain arbitrary code execution through software bugs that would not otherwise be exploitable, such as `NULL` pointer dereferences. To facilitate this, an attacker gains control of the top-level unhandled exception filter for a process in an indirect fashion. While there has been previous work[1, 3] illustrating the usefulness in gaining control of the top-level unhandled exception filter, Microsoft has taken steps in XPSP2 and beyond, such as function pointer encoding[4], to prevent attackers from being able to overwrite and control the unhandled exception filter directly. While this security enhancement is a marked improvement, it is still possible for an attacker to gain control of the top-level unhandled exception filter by taking advantage of a design flaw in the way unhandled exception filters are chained. This approach, however, is limited by an attackers ability to control the chaining of unhandled exception filters, such as through the loading and unloading of DLLs. This does reduce the global impact of this approach; however, there are some interesting cases where it can be immediately applied, such as with Internet Explorer.

Disclaimer: This document was written in the interest of education. The authors cannot be held responsible for how the topics discussed in this document are applied.

Thanks: The authors would like to thank H D Moore, and everyone who learns because it's fun.

Update: This issue has now been addressed by the patch included in MS06-051. A complete analysis has not yet been performed to ensure that it patches all potential vectors.

With that, on with the show...

Chapter 2

Introduction

In the security field, software bugs can be generically grouped into two categories: exploitable or non-exploitable. If a software bug is exploitable, then it can be leveraged to the advantage of the attacker, such as to gain arbitrary code execution. However, if a software bug is non-exploitable, then it is not possible for the attacker to make use of it for anything other than perhaps crashing the application. In more cases than not, software bugs will fall into the category of being non-exploitable simply because they typically deal with common mistakes or invalid assumptions that are not directly related to buffer management or loop constraints. This can be frustrating during auditing and product analysis from an assessment standpoint. With that in mind, it only makes sense to try think of ways to turn otherwise non-exploitable issues into exploitable issues.

In order to accomplish this feat, it's first necessary to try to consider execution vectors that could be redirected to code that the attacker controls after triggering a non-exploitable bug, such as a `NULL` pointer dereference. For starters, it is known that the triggering of a `NULL` pointer dereference will cause an access violation exception to be dispatched. When this occurs, the user-mode exception dispatcher will call the registered exception handlers for the thread that generated the exception, allowing each the opportunity to handle the exception. If none of the exception handlers know what to do with it, the user-mode exception dispatcher will call the top-level *unhandled exception filter* (UEF) via `kernel32!UnhandledExceptionFilter` (if one has been set). The implementation of a function that is set as the registered top-level UEF is not specified, but in most cases it will be designed to pass exceptions that it cannot handle onto the top-level UEF that was registered previously, effectively creating a chain of UEFs. This process will be explained in more detail in the next chapter.

Aside from the exception dispatching process, there are not any other control-

lable execution vectors that an attacker might be able to redirect without some other situation-specific conditions. For that reason, the most important place to look for a point of redirection is within the exception dispatching process itself. This will provide a generic means of gaining execution control for any bug that can be made to crash an application.

Since the first part of the exception dispatching process is the calling of registered exception handlers for the thread, it may make sense to see if there are any controllable execution paths taken by the registered exception handlers at the time that the exception is triggered. This may work in some cases, but is not universal and requires analysis of the specific exception handler routines. Without having an ability to corrupt the list of exception handlers, there is likely to be no other method of redirecting this phase of the exception dispatching process.

If none of the registered exception handlers can be redirected, one must look toward a method that can be used to redirect the unhandled exception filter. This could be accomplished by changing the function pointer to call into controlled code as illustrated in[1, 3]. However, Microsoft has taken steps in XPSP2, such as encoding the function pointer that represents the top-level UEF[4]. This no longer makes it feasible to directly overwrite the global variable that contains the top-level UEF. With that in mind, it may also make sense to look at the function associated with top-level UEF at the time that the exception is dispatched in order to see if the function itself has any meaningful way to redirect its execution.

From this initial analysis, one is left with being required to perform an application-dependent analysis of the registered exception handlers and UEFs that exist at the time that the exception is dispatched. Though this may be useful in some situations, they are likely to be few and far between. For that reason, it makes sense to try to dive one layer deeper to learn more about the exception dispatching process. Chapter 3 will describe in more detail how unhandled exception filters work, setting the stage for the focus of this paper. Based on that understanding, chapter 4 will expound upon an approach that can be used to gain indirect control of the top-level UEF. Finally, chapter 5 will formalize the results of this analysis in an example of a working exploit that takes advantage of one of the many `NULL` pointer dereferences in Internet Explorer to gain arbitrary code execution.

Chapter 3

Understanding Unhandled Exception Filters

This chapter provides an introductory background into the way unhandled exception filters are registered and how the process of filtering an exception that is not handled actually works. This information is intended to act as a base for understanding the attack vector described in chapter 4. If the reader already has sufficient understanding of the way unhandled exception filters operate, feel free to skip ahead.

3.1 Setting the Top-Level UEF

In order to make it possible for applications to handle all exceptions on a process-wide basis, the exception dispatcher exposes an interface for registering an unhandled exception filter. The purpose of the unhandled exception filter is entirely application specific. It can be used to log extra information about an unhandled exception, perform some advanced error recovery, handle language-specific exceptions, or any sort of other task that may need to be taken when an exception occurs that is not handled. To specify a function that should be used as the top-level unhandled exception filter for the process, a call must be made to `kernel32!SetUnhandledExceptionFilter` which is prototyped as[6]:

```
LPTOP_LEVEL_EXCEPTION_FILTER SetUnhandledExceptionFilter(
  LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter
);
```

When called, this function will take the function pointer passed in as the

`lpTopLevelExceptionFilter` argument and encode it using `kernel32!RtlEncodePointer`. The result of the encoding will be stored in the global variable `kernel32!BasepCurrentTopLevelFilter`, thus superseding any previously established top-level filter. The previous value stored within this global variable is decoded using `kernel32!RtlDecodePointer` and returned to the caller. Again, the encoding and decoding of this function pointer is intended to prevent attackers from being able to use an arbitrary memory overwrite to redirect it as has been done pre-XPSP2.

There are two reasons that `kernel32!SetUnhandledExceptionFilter` returns a pointer to the original top-level UEF. First, it makes it possible to restore the original top-level UEF at some point in the future. Second, it makes it possible to create an implicit "chain" of UEFs. In this design, each UEF can make a call down to the previously registered top-level UEF by doing something like the pseudo code below:

```
... app specific handling ...

if (!IsBadCodePtr(PreviousTopLevelUEF))
    return PreviousTopLevelUEF(ExceptionInfo);
else
    return EXCEPTION_CONTINUE_SEARCH;
```

When a block of code that has registered a top-level UEF wishes to deregister itself, it does so by setting the top-level UEF to the value that was returned from its call to `kernel32!SetUnhandledExceptionFilter`. The reason it does it this way is because there is no true list of unhandled exception filters that is maintained. This method of deregistering has one very important property that will serve as the crux of this document. Since deregistration happens in this fashion, the register and deregister operations associated with a top-level UEF *must occur in symmetric order*. An example of this is illustrated in figure 3.1, where top-level UEFs Fx and Gx are registered and deregistered in symmetric order.

In the diagram in figure 3.1, the top-level UEF Fx is registered, returning Nx as the previous top-level UEF. Following that, Gx is registered, returning Fx as the previous value. After some period of time, Gx is deregistered by setting Fx as the top-level UEF, thus returning the top-level UEF to the value it contained before Gx was registered. Finally, Fx deregisters by setting Nx as the top-level UEF.

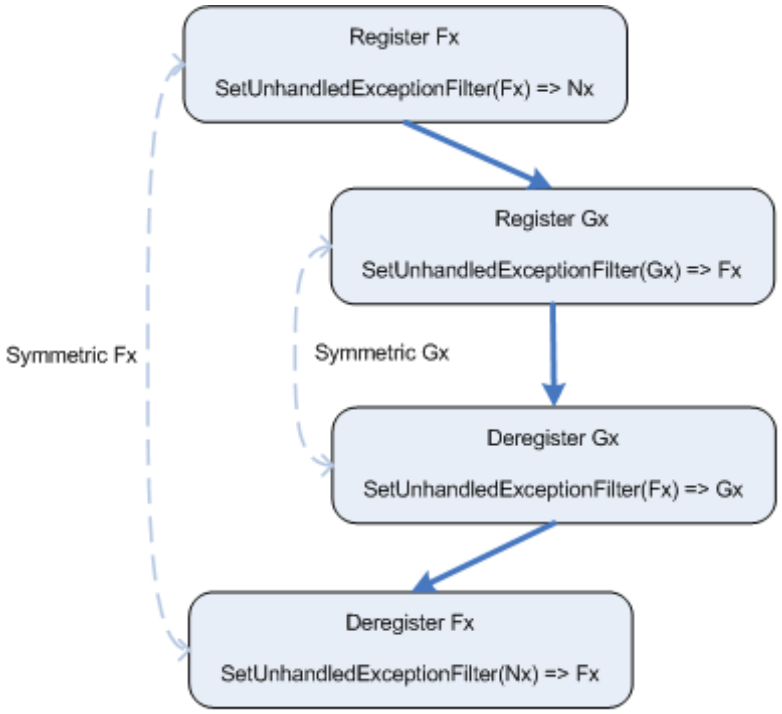

Figure 3.1: Symmetric register and deregister of UEFs

3.2 Handling Unhandled Exceptions

When an exception goes through the initial phase of the exception dispatching process and is not handled by any of the registered exception handlers for the thread that the exception occurred in, the exception dispatcher must take one final stab at getting it handled before forcing the application to terminate. One of the options the exception dispatcher has at this point is to pass the exception to a debugger, assuming one is attached. Otherwise, it has no choice but to try to handle the exception internally and abort the application if that fails. To allow this to happen, applications can make a call to the unhandled exception filter associated with the process as described in [5]. In the general case, calling the unhandled exception filter will result in `kernel32!UnhandledExceptionFilter` being called with information about the exception being dispatched.

The job of `kernel32!UnhandledExceptionFilter` is two fold. First, if a debugger is not present, it must make a call to the top-level UEF registered with the process. The top-level UEF can then attempt to handle the exception, possibly recovering and allowing execution to continue, such as by returning

EXCEPTION_CONTINUE_EXECUTION. Failing that, it can either forcefully terminate the process, typically by returning EXCEPTION_EXECUTE_HANDLER or allow the normal error reporting dialog to be displayed by returning EXCEPTION_CONTINUE_SEARCH. If a debugger is present, the unhandled exception filter will attempt to pass the exception on to the debugger in order to give it a chance to handle the exception. When this occurs, the top-level UEF is *not* called. This is important to remember as the paper goes on, as it can be a source of trouble if one forgets this fact.

When operating with no debugger present, kernel32!UnhandledExceptionFilter will attempt to decode the function pointer associated with the top-level UEF by calling kernel32!RtlDecodePointer on the global variable that contains the top-level UEF, kernel32!kernel32!BasepCurrentTopLevelFilter, as shown below:

```
7c862cc1 ff35ac33887c push dword ptr [kernel32!BasepCurrentTopLevelFilter]
7c862cc7 e8e1d6faff   call kernel32!RtlDecodePointer (7c8103ad)
```

If the value returned from kernel32!RtlDecodePointer is not NULL, then a call is made to the now-decoded top-level UEF function, passing the exception information on:

```
7c862ccc 3bc7         cmp     eax,edi
7c862cce 7415         jz kernel32!UnhandledExceptionFilter+0x15b (7c862ce5)
7c862cd0 53           push    ebx
7c862cd1 ffd0         call    eax
```

The return value of the filter will control whether or not the application continues execution, terminates, or reports an error and terminates.

3.3 Uses for Unhandled Exception Filters

In most cases, unhandled exception filters are used for language-specific exception handling. This usage is all done transparently to programmers of the language. For instance, C++ code will typically register an unhandled exception filter through CxxSetUnhandledExceptionFilter during CRT initialization as called from the entry point associated with the program or shared library. Likewise, C++ will typically deregister the unhandled exception filter that it registers by calling CxxRestoreUnhandledExceptionFilter during program termination or shared library unloading.

Other uses include programs that wish to do advanced error reporting or information collection prior to allowing an application to terminate due to an unhandled exception.

Chapter 4

Gaining Control of the Unhandled Exception Filter

At this point, the only feasible vector for gaining control of the top-level UEF is to cause calls to be made to `kernel32!SetUnhandledExceptionFilter`. This is primarily due to the fact that the global variable has the current function pointer encoded. One could consider attempting to cause code to be redirected directly to `kernel32!SetUnhandledExceptionFilter`, but doing so would require some kind of otherwise-exploitable vulnerability in an application, thus making it not useful in the context of this document.

Given these restrictions, it makes sense to think a little bit more about the process involved in registering and deregistering UEFs. Since the chain of registered UEFs is implicit, it may be possible to cause that chain to become corrupt or invalid in some way that might be useful. One of the requirements that is known about the registration process for top-level UEFs is that the register and deregister operations must be symmetric. What happens if they aren't, though? Consider the diagram in figure 4.1, where *Fx* and *Gx* are registered and deregistered, but in asymmetric order.

As shown in the diagram in figure 4.1, *Fx* and *Gx* are registered first. Following that, *Fx* is deregistered prior to deregistering *Gx*, thus making the operation asymmetrical. As a result of *Fx* deregistering first, the top-level UEF is set to *Nx*, even though *Gx* should technically still be a part of the chain. Finally, *Gx* deregisters, setting the top-level UEF to *Fx* even though *Fx* had been previously deregistered. This is obviously incorrect behavior, but the code associated with *Gx* has no idea that *Fx* has been deregistered due to the implicit chain that is created.

If asymmetric registration of UEFs can be made to occur, it might be possible

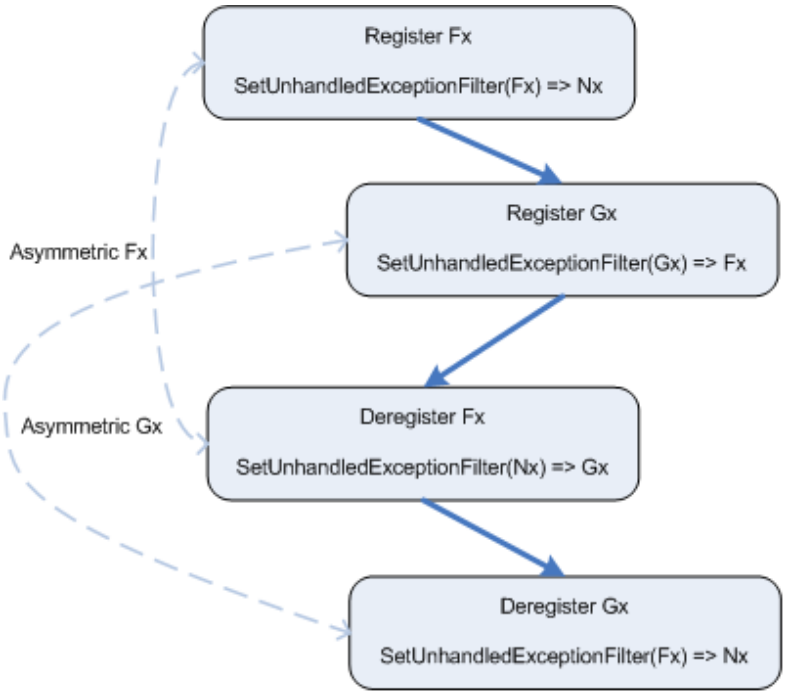

Figure 4.1: Asymmetric register and deregister of UEFs

for an attacker to gain control of the top-level UEF. Consider for a moment that the register and deregister operations in the diagram in figure 4.1 occur during DLL load and unload, respectively. If that is the case, then after deregistration occurs, the DLLs associated with the UEFs will be unloaded. This will leave the top-level UEF set to *Fx* which now points to an invalid region of memory. If an exception occurs after this point and is not handled by a registered exception handler, the unhandled exception filter will be called. If a debugger is not attached, the top-level UEF *Fx* will be called. Since *Fx* points to memory that is no longer associated with the DLL that contained *Fx*, the process will terminate — or worse.

From a security prospective, the act of leaving a dangling function pointer that now points to unallocated memory can be a dream come true. If a scenario such as this occurs, an attacker can attempt to consume enough memory that will allow them to store arbitrary code at the location that the function originally resided. In the event that the function is called, the attacker's arbitrary code will be executed rather than the code that was was originally at that location. In the case of the top-level UEF, the only thing that an attacker would need to do in order to cause the function pointer to be called is to generate an unhandled exception, such as a `NULL` pointer dereference.

All of these details combine to provide a feasible vector for executing arbitrary code. First, it's necessary to be able to cause at least two DLLs that set UEFs to be deregistered asymmetrically, thus leaving the top-level UEF pointing to invalid memory. Second, it's necessary to consume enough memory that attacker controlled code can reside at the location that one of the UEF functions originally resided. Finally, an exception must be generated that causes the top-level UEF to be called, thus executing the attacker's arbitrary code.

The big question, though, is how feasible is it to really be able to control the registering and deregistering of UEFs? To answer that, chapter 5 provides a case study on one such application where it's all too possible: Internet Explorer.

Chapter 5

Case Study: Internet Explorer

Unfortunately for Internet Explorer, it's time for it to once again dawn the all-too-exploitable hat and tell us about how it can be used as a medium to gain arbitrary code execution with all otherwise non-exploitable bugs. In this approach, Internet Explorer is used as a medium for causing DLLs that register and deregister top-level UEFs to be loaded and unloaded. One way in which an attacker can accomplish this is by using Internet Explorer's facilities for instantiating COM objects from within the browser. This can be accomplished either by using the `new ActiveXObject` construct in JavaScript or by using the HTML `OBJECT` tag.

In either case, when a COM object is being instantiated, the DLL associated with that COM object will be loaded into memory if the object instance is created using the `INPROC_SERVER`. When this happens, the COM object's `DllMain` will be called. If the DLL has an unhandled exception filter, it may be registered during CRT initialization as called from the DLL's entry point. This takes care of the registering of UEFs, so long as COM objects that are associated with DLLs that set UEFs can be found.

To control the deregister phase, it is necessary to somehow cause the DLLs associated with the previously instantiated COM objects to be unloaded. One approach that can be taken to do this is attempt to leverage the locations that `ole32!CoFreeUnusedLibrariesEx` is called from. One particular place that it's called from is during the closure of an Internet Explorer window that once hosted the COM object. When this function is called, all currently loaded COM DLLs will have their `DllCanUnloadNow` routines called. If the routine returns `S_OK`, such as when there are no outstanding references to COM objects hosted by the DLL, then the DLL can be unloaded.

Now that techniques for controlling the loading and unloading of DLLs that set UEFs has been identified, it's necessary to come up with an implementation that will allow the deregister phase to occur asymmetrically. One method that can be used to accomplish this illustrated by the registration phase in figure 5.1 and the deregistration phase in figure 5.2.

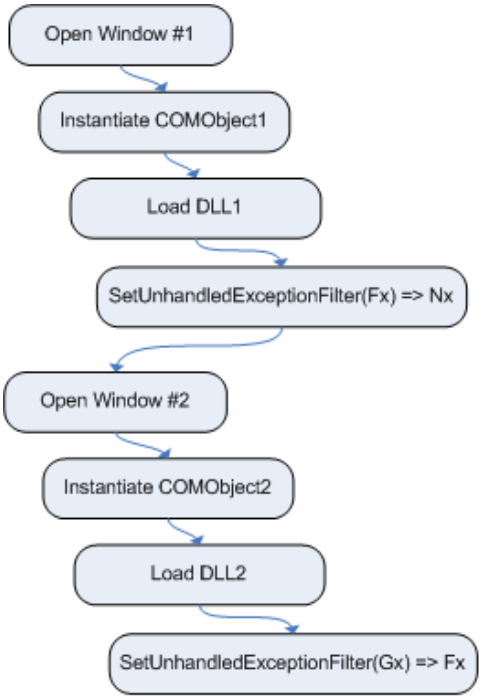

Figure 5.1: Registering Top-Level UEFs through COM Objects

In the example described in figure 5.1, two windows are opened, each of which registers a UEF by way of a DLL that implements a specific COM object. In this example, the first window instantiates `COMObject1` which is implemented by DLL #1. When DLL #1 is loaded, it registers a top-level UEF *Fx*. Once that completes, the second window is opened which instantiates `COMObject2`, thus causing DLL #2 to be loaded which also registers a top-level UEF, *Gx*. Once these operations complete, DLL #1 and DLL #2 are still resident in memory and the top-level UEF points to *Gx*.

To gain control of the top-level UEF, *Fx* and *Gx* will need to be deregistered asymmetrically. To accomplish this, DLL #1 must be unloaded before DLL #2. This can be done by closing the window that hosts `COMObject1`, thus causing `ole32!CoFreeUnusedLibrariesEx` to be called which results in DLL #1 being unloaded. Following that, the window that hosts `COMObject2` should be closed,

once again causing unused libraries to be freed and DLL #2 unloaded. The diagram in figure 5.2 illustrates this process.

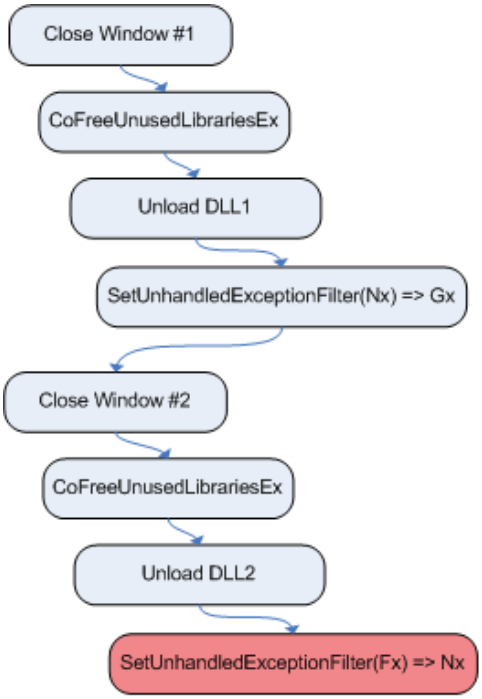

Figure 5.2: Deregistering Top-Level UEFs through COM Objects Asymmetrically

After the process in figure 5.2 completes, *Fx* will be the top-level UEF for the process, even though the DLL that hosts it, DLL #1, has been unloaded. If an exception occurs at this point in time, the unhandled exception filter will make a call to a function that now points to an invalid region of memory.

At this point, an attacker now has reasonable control over the top-level UEF but is still in need of some approach that can used to place his or her code at the location that *Fx* resided at. To accomplish this, attackers can make use of the heap-spraying[8, 7] technique that has been commonly applied to browser-based vulnerabilities. The purpose of the heap-spraying technique is to consume an arbitrary amount of memory that results in the contents of the heap growing toward a specific address region. The contents, or spray data, is arbitrary code that will result in an attacker's direct or indirect control of execution flow once the vulnerability is triggered. For the purpose of this paper, the trigger is the generation of an arbitrary exception.

As stated above, the heap-spraying technique can be used to place code at

the location that *Fx* resided. However, this is limited by whether or not that location is close enough to the heap to be a practical target for heap-spraying. In particular, if the heap is growing from `0x00480000` and the DLL that contains *Fx* was loaded at `0x7c800000`, it would be a requirement that roughly *1.988 GB* of data be placed in the heap. That is, of course, assuming that the target machine has enough memory to contain this allocation (across RAM and swap). Not to mention the fact that spraying that much data could take an inordinate amount of time depending on the speed of the machine. For these reasons, it is typically necessary for the DLL that contains *Fx* in this example scenario to be mapped at an address that is as close as possible to a region that the heap is growing from.

During the research of this attack vector, it was found that all of the COM DLLs provided by Microsoft on XPSP2 are compiled to load at higher addresses which make them challenging to reach with heap-spraying, but it's not impossible. Many 3rd party COM DLLs, however, are compiled with a default load address of `0x00400000`, thus making them perfect candidates for this technique. Another thing to keep in mind is that the preferred load address of a DLL is just that: preferred. If two DLLs have the same preferred load address, or their mappings would overlap, then obviously one would be relocated to a new location, typically at a lower address close to the heap, when it is loaded. By keeping this fact in mind, it may be possible to load DLLs that overlap, forcing relocation of a DLL that sets a UEF that would otherwise be loaded at a higher address.

It is also very important to note that a COM object does not have to be successfully instantiated for the DLL associated with it to be loaded into memory. This is because in order for Internet Explorer to determine whether or not the COM class can be created and is compatible with one that may be used from Internet Explorer, it must load and query various COM interfaces associated with the COM class. This fact is very useful because it means that any DLL that hosts a COM object can be used — not just ones that host COM objects that can be successfully instantiated from Internet Explorer.

The culmination of all of these facts is a functional proof of concept exploit for Windows XP SP2 and the latest version of Internet Explorer with all patches applied prior to MS06-051. Its one requirement is that the target have Adobe Acrobat installed. Alternatively, other 3rd party (or even MS provided DLLs) can be used so long as they can be feasibly reached with heap-spraying techniques. Technically speaking, this proof of concept exploits a `NULL` pointer dereference to gain arbitrary code execution. It has been implemented as an exploit module for the 3.0 version of the Metasploit Framework.

The following example shows this proof of concept in action:

```
msf exploit(windows/browser/ie_unexpfilt_poc) > exploit
[*] Started reverse handler
```

```
[*] Using URL: http://x.x.x.x:8080/FnhWjeVOnU8NlbAGAEhjcjzQWh17myEK1Exg0
[*] Server started.
[*] Exploit running as background job.
msf exploit(windows/browser/ie_unexpfilt_poc) >
[*] Sending stage (474 bytes)
[*] Command shell session 1 opened (x.x.x.x:4444 -> y.y.y.y:1059)

msf exploit(windows/browser/ie_unexpfilt_poc) > session -i 1
[*] Starting interaction with 1...

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\mmiller\Desktop>
```

Chapter 6

Mitigation Techniques

In the interest of not presenting a problem without a solution, the authors have devised a few different approaches that might be taken by Microsoft to solve this issue. Prior to identifying the solution, it is important to summarize the root of the problem. In this case, the authors feel that the problem at hand is rooted around a design flaw with the way the unhandled exception filter "chain" is maintained. In particular, the "chain" management is an implicit thing which hinges on the symmetric registering and deregistering of unhandled exception filters. In order to solve this design problem, some mechanism must be put in place that will eliminate the symmetrical requirement. Alternatively, the symmetrical requirement could be retained so long as something ensured that operations never occurred out of order. The authors feel that this latter approach is more complicated and potentially not feasible. The following sections will describe a few different approaches that might be used or considered to solve this issue.

Aside from architecting a more robust implementation, this attack vector may also be mitigated through conventional exploitation counter-measures, such as NX and ASLR.

6.1 Behavioral Change to SetUnhandledExceptionFilter

One way in which Microsoft could solve this issue would be to change the behavior of `kernel32!SetUnhandledExceptionFilter` in a manner that allows it to support true registration and deregistration operations rather than implicit ones. This can be accomplished by making it possible for the function to determine whether a register operation is occurring or whether a deregister operation

is occurring.

Under this model, when a registration operation occurs, `kernel32!SetUnhandledExceptionFilter` can return a dynamically generated context that merely calls the routine that is previous to the one that was registered. The fact that the context is dynamically generated makes it possible for the function to distinguish between registrations and deregistrations. When the function is called with a dynamically generated context, it can assume that a deregistration operation os occurring. Otherwise, it must assume that a registration operation is occurring.

To ensure that the underlying list of registered UEFs is not corrupted, `kernel32!SetUnhandledExceptionFilter` can be modified to ensure that when a deregistration operation occurs, any dynamically generated contexts that reference the routine being deregistered can be updated to call to the next-previous routine, if any, or simply return if there is no longer a previous routine.

6.2 Prevent Setting of non-image UEF

One approach that could be used to solve this issue for the general case is the modification of `kernel32!SetUnhandledExceptionFilter` to ensure that the function pointer being passed in is associated with an image region. By adding this check at the time this function is called, the attack vector described in this document can be mitigated. However, doing it in this manner may have negative implications for backward compatibility. For instance, there are likely to be cases where this scenario happens completely legitimately without malicious intent. If a check like this were to be added, a once-working application would begin to fail due to the added security checks. This is not an unlikely scenario. Just because an unhandled exception filter is is invalid doesn't mean that it will eventually cause the application to crash because it may, in fact, never be executed.

6.3 Prevent Execution of non-image UEF

Like preventing the setting of a non-image UEF, it may also be possible to to modify `kernel32!UnhandledExceptionFilter` to prevent execution of the top-level UEF if it points to a non-image region. While this seems like it would be a useful check and should solve the issue, the fact is that it does not. Consider the scenario where a top-level UEF is set to an invalid address due to asymmetric deregistration. Following that, the top-level UEF is set to a new value which is the location of a valid function. After this point, if an unhandled exception is dispatched, `kernel32!UnhandledExceptionFilter` will see that the top-level UEF points to a valid image region and as such will call it. However, the top-

level UEF may be implemented in such a way that it will pass exceptions that it cannot handle onto the previously registered top-level UEF. When this occurs, the invalid UEF is called which may point to arbitrary code at the time that it's executed. The fact that `kernel32!UnhandledExceptionFilter` can filter non-image regions does not solve the fact that uncontrolled UEFs may pass exceptions on up the chain.

Chapter 7

Future Research

With the technique identified for being able to control the top-level UEF by taking advantage of asymmetric deregistration, future research can begin to identify better ways in which to accomplish this. For instance, rather than relying on child windows in Internet Explorer, there may be another vector through which `ole32!CoFreeUnusuedLibrariesEx` can be called to cause the asymmetric deregistration to occur[1]. There may also be better and more refined techniques that can be used to more accurately spray the heap in order to place arbitrary code at the location that a defunct top-level UEF resided at.

Aside from improving the technique itself, it is also prudent to consider other software applications this could be affected by this. In most cases, this technique will not be feasible due to an attacker's inability to control the loading and unloading of DLLs. However, should a mechanism for accomplishing this be exposed, it may indeed be possible to take advantage of this.

One such target software application that the authors find most intriguing would be IIS. If it were possible for a remote attacker to cause DLLs that use UEFs to be loaded and unloaded in a particular order, such as by accessing websites that load COM objects, then it may be possible for an attacker to leverage this vector on a remote webserver. At the time of this writing, the only approach that the authors are aware of that could permit this are remote debugging features present in ASP.NET that allow for the instantiation of COM objects that are placed in a specific allow list. This isn't a very common configuration, and is also limited by which COM objects can be instantiated, thus making it not particularly feasible. However, it is thought that other, more feasible techniques may exist to accomplish this.

Aside from IIS, the authors are also of the opinion that this attack vector could

[1] By default, `ole32!CoFreeUnusedLibrariesEx` is called every ten minutes, but this fact is not particulary useful in terms of general exploitation

be applied to many of the Microsoft Office applications, such as Excel and Word. These suites are thought to be vulnerable due to the fact that they permit the instantiation and embedding of arbitrary COM objects in the document files. If it were possible to come up with a way to control the loading and unloading of DLLs through these instantiations, it may be possible to take advantage of the flaw outlined in this paper. One particular way in which this may be possible is through the use of macros, but this has a lesser severity because it would require some form of user interaction to permit the execution of macros.

Another interesting application that may be susceptible to this attack is Microsoft SQL server. Due to the fact that SQL server has features that permit the loading and unloading of DLLs[2], it may be possible to leverage a SQL injection attack in a way that makes it possible to gain control of the top-level UEF by causing certain DLLs to be loaded and unloaded[2]. Once that occurs, a large query with predictable results could be used as a mechanism to spray the heap. This type of attack could even be accomplished through something as innocuous as a website that is merely backed against the SQL server. Remember, attack vectors aren't always direct.

[2]However, given the ability to load DLLs, there are likely to be other techniques that can be used to gain code execution as well

Chapter 8

Conclusion

The title of this paper implies that an attacker has the ability to leverage code execution of bugs that would otherwise not be useful, such as `NULL` pointer dereferences. To that point, this paper has illustrated a technique that can be used to gain control of the top-level unhandled exception filter for an application by making the registration and deregistration process asymmetrical. Once the top-level UEF has been made to point to invalid memory, an attacker can use techniques like heap-spraying to attempt to place attacker controlled code at the location that the now-defunct top-level UEF resided at. Assuming this can be accomplished, an attacker simply needs to be able to trigger an unhandled exception to cause the execution of arbitrary code.

The crux of this attack vector is in leveraging a design flaw in the assumptions made by the way the unhandled exception filter "chain" is maintained. In particular, the design assumes that calls made to register, and subsequently deregister, an unhandled exception filter through `kernel32!SetUnhandledExceptionFilter` will be done symmetrically. However, this cannot always be controlled, as DLLs that register unhandled exception filters are not always guaranteed to be loaded and unloaded in a symmetric fashion. If an attacker is capable of controlling the order in which DLLs are loaded and unloaded, then they may be capable of gaining arbitrary code execution through this technique, such as was illustrated in the Internet Explorer case study in chapter 5.

While not feasible in most cases, this technique has been proven to work in at least one critical application: Internet Explorer. Going forward, other applications, such as IIS, may also be found to be susceptible to this attack vector. All it will take is a little creativity and the right set of conditions.

Bibliography

[1] Conover, Matt and Oded Horovitz. *Reliable Windows Heap Exploits*. http://cansecwest.com/csw04/csw04-Oded+Connover.ppt; accessed May 6, 2006.

[2] Kazienko, Przemyslaw and Piotr Dorosz. *Hacking an SQL Server*. http://www.windowsecurity.com/articles/Hacking_an_SQL_Server.html; accessed May 7, 2006.

[3] Litchfield, David. *Windows Heap Overflows*. http://www.blackhat.com/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.ppt; accessed May 6, 2006.

[4] Howard, Michael. *Protecting against Pointer Subterfuge (Kinda!)*. http://blogs.msdn.com/michael_howard/archive/2006/01/30/520200.aspx; accessed May 6, 2006.

[5] Microsoft Corporation. *UnhandledExceptionFilter*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/unhandledexceptionfilter.asp; accessed May 6, 2006.

[6] Microsoft Corporation. *SetUnhandledExceptionFilter*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/setunhandledexceptionfilter.asp; accessed May 6, 2006.

[7] Murphy, Matthew. *Windows Media Player Plug-In Embed Overflow*; http://www.milw0rm.com/exploits/1505; accessed May 7, 2006.

[8] SkyLined. *InternetExploiter*. http://www.edup.tudelft.nl/~bjwever/exploits/InternetExploiter2.zip; accessed May 7, 2006.