Preventing the Exploitation of SEH Overwrites

9/2006

skape
mmiller@hick.org

Contents

Chapter 1

Foreword

Abstract: This paper proposes a technique that can be used to prevent the exploitation of SEH overwrites on 32-bit Windows applications without requiring any recompilation. While Microsoft has attempted to address this attack vector through changes to the exception dispatcher and through enhanced compiler support, such as with `/SAFESEH` and `/GS`, the majority of benefits they offer are limited to image files that have been compiled to make use of the compiler enhancements. This limitation means that without all image files being compiled with these enhancements, it may still be possible to leverage an SEH overwrite to gain code execution. In particular, many third-party applications are still vulnerable to SEH overwrites even on the latest versions of Windows because they have not been recompiled to incorporate these enhancements. To that point, the technique described in this paper does not rely on any compile time support and instead can be applied at runtime to existing applications without any noticeable performance degradation. This technique is also backward compatible with all versions of Windows NT+, thus making it a viable and proactive solution for legacy installations.

Thanks: The author would like to thank all of the people who have helped with offering feedback and ideas on this technique. In particular, the author would like to thank spoonm, H D Moore, Skywing, Richard Johnson, and Alexander Sotirov.

Chapter 2

Introduction

Like other operating systems, the Windows operating system finds itself vulnerable to the same classes of vulnerabilities that affect other platforms, such as stack-based buffer overflows and heap-based buffer overflows. Where the platforms differ is in terms of how these vulnerabilities can be leveraged to gain code execution. In the case of a conventional stack-based buffer overflow, the overwriting of the return address is the most obvious and universal approach. However, unlike other platforms, the Windows platform has a unique vector that can, in many cases, be used to gain code execution through a stack-based overflow that is more reliable than overwriting the return address. This vector is known as a *Structured Exception Handler* (SEH) overwrite. This attack vector was publicly discussed for the first time, as far as the author is aware, by David Litchfield in his paper entitled *Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server*[2][1].

In order to completely understand how to go about protecting against SEH overwrites, it's prudent to first spend some time describing the intention of the facility itself and how it can be abused to gain code execution. To provide this background information, a description of structured exception handling will be given in section 2.1. Section 2.2 provides an illustration of how an SEH overwrite can be used to gain code execution. If the reader already understands how structured exception handling works and can be exploited, feel free to skip ahead. The design of the technique that is the focus of this paper will be described in chapter 3 followed by a description of a proof of concept implementation in chapter 4. Finally, potential compatibility issues are noted in chapter 5.

[1] However, exploits had been using this technique prior to the publication, so it is unclear who originally found the technique

2.1 Structured Exception Handling

Structured Exception Handling (SEH) is a uninform system for dispatching and handling exceptions that occur during the normal course of a program's execution. This system is similar in spirit to the way that UNIX derivatives use signals to dispatch and handle exceptions, such as through `SIGPIPE` and `SIGSEGV`. SEH, however, is a more generalized and powerful system for accomplishing this task, in the author's opinion. Microsoft's integration of SEH spans both user-mode and kernel-mode and is a licensed implementation of what is described in a patent owned by Borland[1]. In fact, this patent is one of the reasons why open source operating systems have not chosen to integrate this style of exception dispatching[11].

In terms of implementation, structured exception handling works by defining a uniform way of handling all exceptions that occur during the normal course of process execution. In this context, an exception is defined as an event that occurs during execution that necessitates some form of extended handling. There are two primary types of exceptions. The first type, known as a *hardware* exception, is used to categorize exceptions that originate from hardware. For example, when a program makes reference to an invalid memory address, the processor will raise an exception through an interrupt that gives the operating system an opportunity to handle the error. Other examples of hardware exceptions include illegal instructions, alignment faults, and other architecture-specific issues. The second type of exception is known as a *software* exception. A software exception, as one might expect, originates from software rather than from the hardware. For example, in the event that a process attempts to close an invalid handle, the operating system may generate an exception.

One of the reasons that the word structured is included in structured exception handling is because of the fact that it is used to dispatch both hardware and software exceptions. This generalization makes it possible for applications to handle all types of exceptions using a common system, thus allowing for greater application flexibility when it comes to error handling.

The most important detail of SEH, insofar as it pertains to this document, is the mechanism through which applications can dynamically register handlers to be called when various types of exceptions occur. The act of registering an exception handler is most easily described as inserting a function pointer into a chain of function pointers that are called whenever an exception occurs. Each exception handler in the chain is given the opportunity to either handle the exception or pass it on to the next exception handler.

At a higher level, the majority of compiler-generated C/C++ functions will register exception handlers in their prologue and remove them in their epilogue. In this way, the exception handler chain mirrors the structure of a thread's stack in that they are both LIFOs (*last-in-first-out*). The exception handler that was

registered last will be the first to be removed from the chain, much the same as last function to be called will be the first to be returned from.

To understand how the process of registering an exception handler actually works in practice, it makes sense to analyze code that makes use of exception handling. For instance, the code below illustrates what would be required to catch all exceptions and then display the type of exception that occurred:

```
__try
{
    ...
} __except(EXCEPTION_EXECUTE_HANDLER)
{
    printf("Exception code: %.8x\n", GetExceptionCode());
}
```

In the event that an exception occurs from code inside of the `__try` / `__except` block, the `printf` call will be issued and `GetExceptionCode` will return the actual exception that occurred. For instance, if code made reference to an invalid memory address, the exception code would be `0xc0000005`, or `EXCEPTION_ACCESS_VIOLATION`. To completely understand how this works, it is necessary to dive deeper and take a look at the assembly that is generated from the C code described above. When disassembled, the code looks something like what is shown below:

```
00401000 55              push    ebp
00401001 8bec            mov     ebp,esp
00401003 6aff            push    0xff
00401005 6818714000      push    0x407118
0040100a 68a4114000      push    0x4011a4
0040100f 64a100000000    mov     eax,fs:[00000000]
00401015 50              push    eax
00401016 64892500000000  mov     fs:[00000000],esp
0040101d 83c4f4          add     esp,0xfffffff4
00401020 53              push    ebx
00401021 56              push    esi
00401022 57              push    edi
00401023 8965e8          mov     [ebp-0x18],esp
00401026 c745fc00000000  mov     dword ptr [ebp-0x4],0x0
0040102d c6050000000001  mov     byte ptr [00000000],0x1
00401034 c745fcffffffff  mov     dword ptr [ebp-0x4],0xffffffff
0040103b eb2b            jmp     ex!main+0x68 (00401068)
0040103d 8b45ec          mov     eax,[ebp-0x14]
00401040 8b08            mov     ecx,[eax]
00401042 8b11            mov     edx,[ecx]
00401044 8955e4          mov     [ebp-0x1c],edx
```

```
00401047 b801000000       mov     eax,0x1
0040104c c3               ret

0040104d 8b65e8           mov     esp,[ebp-0x18]
00401050 8b45e4           mov     eax,[ebp-0x1c]
00401053 50               push    eax
00401054 6830804000       push    0x408030
00401059 e81b000000       call    ex!printf (00401079)
0040105e 83c408           add     esp,0x8
00401061 c745fcffffffff   mov     dword ptr [ebp-0x4],0xffffffff
00401068 8b4df0           mov     ecx,[ebp-0x10]
0040106b 64890d00000000   mov     fs:[00000000],ecx
00401072 5f               pop     edi
00401073 5e               pop     esi
00401074 5b               pop     ebx
00401075 8be5             mov     esp,ebp
00401077 5d               pop     ebp
00401078 c3               ret
```

The actual registration of the exception handler all occurs behind the scenes in the C code. However, in the assembly code, the registration of the exception handler starts at `0x0040100a` and spans four instructions. It is these four instructions that are responsible for registering the exception handler for the calling thread. The way that this actually works is by chaining an `EXCEPTION_REGISTRATION_RECORD` to the front of the list of exception handlers. The head of the list of already registered exception handlers is found in the `ExceptionList` attribute of the `NT_TIB` structure. If no exception handlers are registered, this value will be set to `0xffffffff`. The `NT_TIB` structure makes up the first part of the `TEB`, or *Thread Environment Block*, which is an undocumented structure used internally by Windows to keep track of per-thread state in user-mode. A thread's `TEB` can be accessed in a position-independent fashion by referencing addresses relative to the `fs` segment register. For example, the head of the exception list chain be be obtained through `fs:[0]`.

To make sense of the four assembly instructions that register the custom exception handler, each of the four instructions will be described individually. For reference purposes, the layout of the `EXCEPTION_REGISTRATION_RECORD` is described below:

```
+0x000 Next             : Ptr32 _EXCEPTION_REGISTRATION_RECORD
+0x004 Handler          : Ptr32
```

1. `push 0x4011a4`

   The first instruction pushes the address of the CRT generated `_except_handler3` symbol. This routine is responsible for dispatching general exceptions that

are registered through the __except compiler intrinsic. The key thing to note here is that the virtual address of a function is pushed onto the stack that is excepted to be referenced in the event that an exception is thrown. This push operation is the first step in dynamically constructing an EXCEPTION_REGISTRATION_RECORD on the stack by first setting the Handler attribute.

2. `mov eax,fs:[00000000]`

   The second instruction takes the current pointer to the first EXCEPTION_REGISTRATION_RECORD and stores it in eax.

3. `push eax`

   The third instruction takes the pointer to the first exception registration record in the exception list and pushes it onto the stack. This, in turn, sets the Next attribute of the record that is being dynamically generated on the stack. Once this instruction completes, a populated EXCEPTION_REGISTRATION_RECORD will exist on the stack that takes the following form:

```
+0x000 Next             : 0x0012ffb0
+0x004 Handler          : 0x004011a4     ex!_except_handler3+0
```

4. `mov fs:[00000000],esp`

   Finally, the dynamically generated exception registration record is stored as the first exception registration record in the list for the current thread. This completes the process of inserting a new registration record into the chain of exception handlers.

The important things to take away from this description of exception handler registration are as follows. First, the registration of exception handlers is a runtime operation. This means that whenever a function is entered that makes use of an exception handler, it must dynamically register the exception handler. This has implications as it relates to performance overhead. Second, the list of registered exception handlers is stored on a per-thread basis. This makes sense because threads are considered isolated units of execution and therefore exception handlers are only relative to a particular thread. The final, and perhaps most important, thing to take away from this is that the assembly generated by the compiler to register an exception handler at runtime makes use of the current thread's stack. This fact will be revisited later in this section.

In the event that an exception occurs during the course of normal execution, the operating system will step in and take the necessary steps to dispatch the exception. In the event that the exception occurred in the context of a thread that is running in user-mode, the kernel will take the exception information and generate an EXCEPTION_RECORD that is used to encapsulate all

of the exception information. Furthermore, a snapshot of the executing state of the thread is created in the form of a populated `CONTEXT` structure. The kernel then passes this information off to the user-mode thread by transferring execution from the location that the fault occurred at to the address of `ntdll!KiUserExceptionDispatcher`. The important thing to understand about this is that execution of the exception dispatcher occurs in the context of the thread that generated the exception.

The job of `ntdll!KiUserExceptionDispatcher` is, as the name implies, to dispatch user-mode exceptions. As one might guess, the way that it goes about doing this is by walking the chain of registered exception handlers stored relative to the current thread. The diagram in figure 2.1 provides a basic example of how it walks the chain. As the exception dispatcher walks the chain, it calls the handler associated with each registration record, giving that handler the opportunity to handle, fail, or pass on the exception.

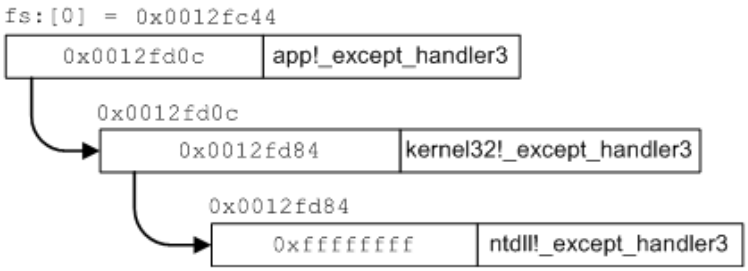

Figure 2.1: Walking the chain of exception registration records

While there are other things involved in the exception dispatching process, this description will suffice to set the stage for how it might be abused to gain code execution.

2.2 Gaining Code Execution

There is one important thing to remember when it comes to trying to gain code execution through an SEH overwrite. Put simply, the fact that each exception registration record is stored on the stack lends itself well to abuse when considered in conjunction with a conventional stack-based buffer overflow. As described in section 2.1, each exception registration record is composed of a `Next` pointer and a `Handler` function pointer. Of most interest in terms of exploitation is the `Handler` attribute. Since the exception dispatcher makes use of this attribute as a function pointer, it makes sense that should this attribute be overwritten with attacker controlled data, it would be possible to gain code

execution. In fact, that's exactly what happens, but with an added catch.

While typical stack-based buffer overflows work by overwriting the return address, an SEH overwrite works by overwriting the `Handler` attribute of an exception registration record that has been stored on the stack. Unlike overwriting the return address, where control is gained immediately upon return from the function, an SEH overwrite does not actually gain code execution until after an exception has been generated. The exception is necessary in order to cause the exception dispatcher to call the overwritten `Handler`.

While this may seem like something of a nuisance that would make SEH overwrites harder to exploit, it's not. Generating an exception that leads to the calling of the `Handler` is as simple as overwriting the return address with an invalid address in most cases. When the function returns, it attempts to execute code from an invalid memory address which generates an access violation exception. This exception is then passed onto the exception dispatcher which calls the overwritten `Handler`.

The obvious question to ask at this point is what benefit SEH overwrites have over the conventional practice of overwriting the return address. To understand this, it's important to consider one of the common practices employed in Windows-based exploits. On Windows, thread stack addresses tend to change quite frequently between operating system revisions and even across process instances. This differs from most UNIX derivatives where stack addresses are typically predictable across multiple operating system revisions. Due to this fact, most Windows-based exploits will indirectly transfer control into the thread's stack by first bouncing off an instruction that exists somewhere in the address space. This instruction must typically reside at an address that is less prone to change, such as within the code section of a binary. The purpose of this instruction is to transfer control back to the stack in a position-independent fashion. For example, a `jmp esp` instruction might be used. While this approach works perfectly fine, it's limited by whether or not an instruction can be located that is both portable and reliable in terms of the address that it resides at. This is where the benefits of SEH overwrites begin to become clear.

When simply overwriting the return address, an attacker is often limited to a small set of instructions that are not typically common to find at a reliable and portable location in the address space. On the other hand, SEH overwrites have the advantage of being able to use another set of instructions that are far more prevalent in the address space of most every process. This set of instructions is commonly referred to as `pop/pop/ret`. The reason this class of instructions can be used with SEH overwrites and not general stack overflows has to do with the method in which exception handlers are called by the exception dispatcher. To understand this, it is first necessary to know what the specific prototype is for the `Handler` field in the `EXCEPTION_REGISTRATION_RECORD` structure:

```
typedef EXCEPTION_DISPOSITION (*ExceptionHandler)(
```

```
IN EXCEPTION_RECORD ExceptionRecord,
IN PVOID EstablisherFrame,
IN PCONTEXT ContextRecord,
IN PVOID DispatcherContext);
```

The field of most importance is the `EstablisherFrame`. This field actually points to the address of the exception registration record that was pushed onto the stack. It is also located at `[esp+8]` when the `Handler` is called. Therefore, if the `Handler` is overwritten with the address of a `pop/pop/ret` sequence, the result will be that the execution path of the current thread will be transferred to the address of the `Next` attribute for the current exception registration record. While this field would normally hold the address of the next registration record, it instead can hold four bytes of arbitrary code that an attacker can supply when triggering the SEH overwrite. Since there are only four contiguous bytes of memory to work with before hitting the `Handler` field, most attackers will use a simple short jump sequence to jump past the handler and into the attacker controlled code that comes after it. Figure 2.2 illustrates what this might look like after an attacker has overwritten an exception registration record in the manner described above.

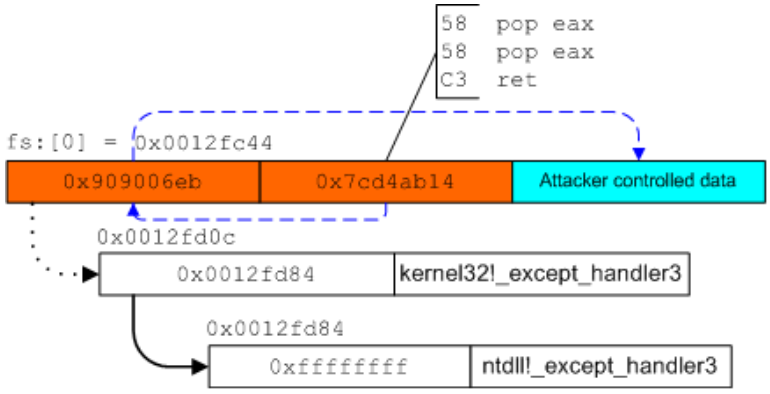

Figure 2.2: Gaining code execution from an SEH overwrite

Chapter 3

Design

The one basic requirement of any solution attempting to prevent the leveraging of SEH overwrites is that it must not be possible for an attacker to be able to supply a value for the `Handler` attribute of an exception registration record that is subsequently used in an unchecked fashion by the exception dispatcher when an exception occurs. If a solution can claim to have satisfied this requirement, then it should be true that the solution is secure.

To that point, Microsoft's solution is secure, but only if all of the images loaded in the address space have been compiled with `/SAFESEH`. Even then, it's possible that it may not be completely secure[1]. If there are any images that have not been compiled with `/SAFESEH`, it may be possible for an attacker to overwrite the `Handler` with an address of an instruction that resides within an unprotected image. The reason Microsoft's implementation cannot protect against this is because SafeSEH works by having the exception dispatcher validate handlers against a table of image-specific safe exception handlers prior to calling an exception handler. Safe exception handlers are stored in a table that is contained in any executable compiled with `/SAFESEH`. Given this limitation, it can also be said that Microsoft's implementation is not secure given the appropriate conditions. In fact, for third-party applications, and even some Microsoft-provided applications, these conditions are considered by the author to be the norm rather than the exception. In the end, it all boils down to the fact that Microsoft's solution is a compile-time solution rather than a runtime solution. With these limitations in mind, it makes sense to attempt to approach the problem from the angle of a runtime solution rather than a compile-time solution.

When it comes to designing a runtime solution, the important consideration that has to be made is that it will be necessary to intercept exceptions before they

[1] For example, it should be possible to overwrite the `Handler` with the address of some non-image associated executable region, if one can be found

are passed off to the registered exception handlers by the exception dispatcher. The particulars of how this can be accomplished will be discussed in chapter 4. Assuming a solution is found to the layering problem, the next step is to come up with a solution for determining whether or not an exception handler is valid and has not been tampered with. While there are many inefficient solutions to this problem, such as coming up with a solution to keep a "secure" list of registered exception handlers, there is one solution in particular that the author feels is bested suited for the problem.

One of the side effects of an SEH overwrite is that the attacker will typically clobber the value of the `Next` attribute associated with the exception registration record that is overwritten. This occurs because the `Next` attribute precedes the `Handler` attribute in memory, and therefore must be overwritten before the `Handler` in the case of a typical buffer overflow. This has a very important side effect that is the key to facilitating the implementation of a runtime solution. In particular, the clobbering of the `Next` attribute means that all subsequent exception registration records would not be reachable by the exception dispatcher when walking the chain.

Consider for the moment a solution that, during thread startup, places a custom exception registration record as the very last exception registration record in the chain. This exception registration record will be symbolically referred to as the *validation frame* henceforth. From that point forward, whenever an exception is about to be dispatched, the solution could walk the chain prior to allowing the exception dispatcher to handle the exception. The purpose of walking the chain before hand is to ensure that the validation frame can be reached. As such, the validation frame's purpose is similar to that of stack canaries[5]. If the validation frame can be reached, then that is evidence of the fact that the chain of exception handlers has not been corrupted. As described above, the act of overwriting the `Handler` attribute also requires that the `Next` pointer be overwritten. If the `Next` pointer is not overwritten with an address that ensures the integrity of the exception handler chain, then this solution can immediately detect that the integrity of the chain is in question and prevent the exception dispatcher from calling the overwritten `Handler`. Figure 3.1 illustrates how this might look at execution time.

Using this technique, the act of ensuring that the integrity of the exception handler chain is kept intact results in the ability to prevent SEH overwrites. The important questions to ask at this point center around what limitations this solution might have. The most obvious question to ask is what's to stop an attacker from simply overwriting the `Next` pointer with the value that was already there. There are a few things that stop this. First of all, it will be common that the attacker does not know the value of the `Next` pointer. Second, and perhaps most important, is that one of the benefits of using an SEH overwrite is that an attacker can make use of a `pop/pop/ret` sequence. By forcing an attacker to retain the value of the `Next` pointer, the major benefit of using an SEH overwrite in the first place is gone. Even conceding this point, an attacker

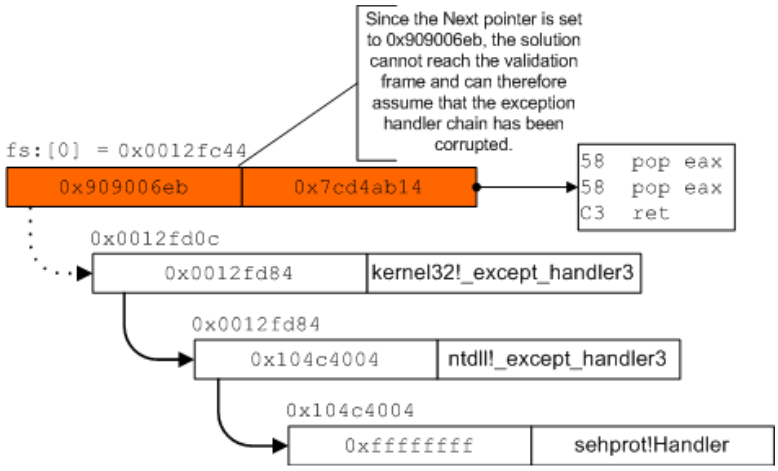

Figure 3.1: Detecting corruption of the exception handler chain

who is able to retain the value of the `Next` pointer would find themselves limited to overwriting the `Handler` with the address of instructions that indirectly transfer control back to their code. However, the attacker won't simply be able to use an instruction like `jmp esp` because the `Handler` will be called in the context of the exception dispatcher. It's at this point that diminishing returns are reached and an attacker is better off simply overwriting the return address, if possible.

Another important question to ask is what's to stop the attacker from overwriting the `Next` pointer with the address of the validation frame itself or, more easily, with `0xffffffff`. The answer to this is much the same as described in the above paragraph. Specifically, by forcing an attacker away from the `pop/pop/ret` sequence, the usefulness of the SEH overwrite vector quickly degrades to the point of it being better to simply overwrite the return address, if possible. However, in order to be sure, the author feels that implementations of this solution would be wise to randomize the location of the validation frame.

It is the author's opinion that the solution described above satisfies the requirement outlined in the beginning of this chapter and therefore qualifies as a secure solution. However, there's always a chance that something has been missed. For that reason, the author is more than happy to be proven wrong on this point.

Chapter 4

Implementation

The implementation of the solution described in the previous chapter relies on intercepting exceptions prior to allowing the native exception dispatcher to handle them such that the exception handler chain can be validated. First and foremost, it is important to identify a way of layering prior to the point that the exception dispatcher transfers control to the registered exception handlers. There are a few different places that this layering could occur at, but the one that is best suited to catch the majority of user-mode exceptions is at the location that `ntdll!KiUserExceptionDispatcher` gains control. However, by hooking `ntdll!KiUserExceptionDispatcher`, it is possible that this implementation may not be able to intercept all cases of an exception being raised, thus making it potentially feasible to bypass the exception handler chain validation.

The best location would be to layer at would be `ntdll!RtlDispatchException`. The reason for this is that exceptions raised through `ntdll!RtlRaiseException`, such as software exceptions, may be passed directly to `ntdll!RtlDispatchException` rather than going through `ntdll!KiUserExceptionDispatcher` first. The condition that controls this is whether or not a debugger is attached to the user-mode process when `ntdll!RtlRaiseException` is called. The reason `ntdll!RtlDispatchException` is not hooked in this implementation is because it is not directly exported. There are, however, fairly reliable techniques that could be used to determine its address. As far as the author is aware, the act of hooking `ntdll!KiUserExceptionDispatcher` should mean that it's only possible to miss software exceptions which are much harder, and in most cases impossible, for an attacker to generate.

In order to layer at `ntdll!KiUserExceptionDispatcher`, the first few instructions of its prologue can be overwritten with an indirect jump to a function that will be responsible for performing any sanity checks necessary. Once the function has completed its sanity checks, it can transfer control back to the original exception dispatcher by executing the overwritten instructions and then jumping

back into `ntdll!KiUserExceptionDispatcher` at the offset of the next instruction to be executed. This is a nice and "clean" way of accomplishing this and the performance overhead is miniscule[1].

In order to hook `ntdll!KiUserExceptionDispatcher`, the first n instructions, where n is the number of instructions that it takes to cover at least 6 bytes, must be copied to a location that will be used by the hook to execute the actual `ntdll!KiUserExceptionDispatcher`. Following that, the first n instructions of `ntdll!KiUserExceptionDispatcher` can then be overwritten with an indirect jump. This indirect jump will be used to transfer control to the function that will validate the exception handler chain prior to allowing the original exception dispatcher to handle the exception.

With the hook installed, the next step is to implement the function that will actually validate the exception handler chain. The basic steps involved in this are to first extract the head of the list from `fs:[0]` and then iterate over each entry in the list. For each entry, the function should validate that the `Next` attribute points to a valid memory location. If it does not, then the chain can be assumed to be corrupt. However, if it does point to valid memory, then the routine should check to see if the `Next` pointer is equal to the address of the validation frame that was previously stored at the end of the exception handler chain for this thread. If it is equal to the validation frame, then the integrity of the chain is confirmed and the exception can be passed to the actual exception dispatcher.

However, if the function reaches an invalid `Next` pointer, or it reaches `0xffffffff` without encountering the validation frame, then it can assume that the exception handler chain is corrupt. It's at this point that the function can take whatever steps are necessary to discard the exception, log that a potential exploitation attempt occurred, and so on. The end result should be the termination of either the thread or the process, depending on circumstances. This algorithm is captured by the pseudo-code below:

```
01: CurrentRecord = fs:[0];
02: ChainCorrupt  = TRUE;
03: while (CurrentRecord != 0xffffffff) {
04:     if (IsInvalidAddress(CurrentRecord->Next))
05:         break;
06:     if (CurrentRecord->Next == ValidationFrame) {
07:         ChainCorrupt = FALSE;
08:         break;
09:     }
10:     CurrentRecord = CurrentRecord->Next;
11: }
12: if (ChainCorrupt == TRUE)
13:     ReportExploitationAttempt();
14: else
15:     CallOriginalKiUserExceptionDispatcher();
```

[1] Where "clean" is defined as the best it can get from a third-party perspective

The above algorithm describes how the exception dispatching path should be handled. However, there is one important part remaining in order to implement this solution. Specifically, there must be some way of registering the validation frame with a thread prior to any exceptions being dispatched on that thread. There are a few ways that this can be accomplished. In terms of a proof of concept, the easiest way of doing this is to implement a DLL that, when loaded into a process' address space, catches the creation notification of new threads through a mechanism like `DllMain` or through the use of a TLS callback in the case of a statically linked library. Both of these approaches provide a location for the solution to establish the validation frame with the thread early on in its execution. However, if there were ever a case where the thread were to raise an exception prior to one of these routines being called, then the solution would improperly detect that the exception handler chain was corrupt.

One solution to this potential problem is to store state relative to each thread that keeps track of whether or not the validation frame has been registered. There are certain implications about doing this, however. First, it could introduce a security problem in that an attacker might be able to bypass the protection by somehow toggling the flag that tracks whether or not the validation frame has been registered. If this flag were to be toggled to no and an exception were generated in the thread, then the solution would have to assume that it can't validate the chain because no validation frame has been installed. Another issue with this is that it would require some location to store this state on a per-thread basis. A good example of a place to store this is in TLS, but again, it has the security implications described above.

A more invasive solution to the problem of registering the validation frame would be to somehow layer very early on in the thread's execution – perhaps even before it begins executing from its entry point. The author is aware of a good way to accomplish this, but it will be left as an exercise to the reader on what this might be. This more invasive solution is something that would be an easy and elegant way for Microsoft to include support for this, should they ever choose to do so.

The final matter of how to go about implementing this solution centers around how it could be deployed and used with existing applications without requiring a recompile. The easiest way to do this in a proof of concept setting would be to implement these protection mechanisms in the form of a DLL that can be dynamically loaded into the address space of a process that is to be protected. Once loaded, the DLL's `DllMain` can take care of getting everything set up. A simple way to cause the DLL to be loaded is through the use of `AppInit_DLLs`[4], although this has some limitations. Alternatively, there are more invasive options that can be considered that will accomplish the goal of loading and initializing the DLL early on in process creation.

One interesting thing about this approach is that while it is targeted at being used as a runtime solution, it can also be used as a compile-time solution.

This means that applications can use this solution at compile-time to protect themselves from SEH overwrites. Unlike Microsoft's solution, this will even protect them in the presence of third-party images that have not been compiled with the support. This can be accomplished through the use of a static library that uses TLS callbacks to receive notifications when threads are created, much like `DllMain` is used for DLL implementations of this solution.

All things considered, the author believes that the implementation described above, for all intents and purposes, is a fairly simplistic way of providing run-time protection against SEH overwrites that has minimal overhead. While the implementation described in this document is considered more suitable for a proof-of-concept or application-specific solution, there are real-world examples of more robust implementations, such as in Wehnus's WehnTrust product[9], a commercial side-project of the author's[2].

[2] Apologies for the shameless plug

Chapter 5

Compatibility

Like most security solutions, there are always compatibility problems that must be considered. As it relates to the solution described in this paper, there are a couple of important things to keep in mind.

The first compatibility issue that might happen in the real world is a scenario where an application invalidates the exception handler chain in a legitimate fashion. The author is not currently aware of situations where an application would legitimately need to do this, but it has been observed that some applications, such as cygwin, will do funny things with the exception handler chain that are not likely to play nice with this form of protection. In the event that an application invalidates the exception handler chain, the solution described in this paper may inadvertently detect that an SEH overwrite has occurred simply because it is no longer able to reach the validation frame.

Another compatibility issue that may occur centers around the fact that the implementation described in this paper relies on the hooking of functions. In almost every situation it is a bad idea to use function hooking, but there are often situations where there is no alternative, especially in closed source environments. The use of function hooking can lead to compatibility problems with other applications that also hook `ntdll!KiUserExceptionDispatcher`. There may also be instances of security products that detect the hooking of `ntdll!KiUserExceptionDispatcher` and classify it as malware-like behavior. In any case, these compatibility concerns center less around the fundamental concept and more around the specific implementation that would be required of a third-party.

Chapter 6

Conclusion

Software-based vulnerabilities are a common problem that affect a wide array of operating systems. In some cases, these vulnerabilities can be exploited with greater ease depending on operating system specific features. One particular case of where this is possible is through the use of an SEH overwrite on 32-bit applications on the Windows platform. An SEH overwrite involves overwriting the `Handler` associated with an exception registration record. Once this occurs, an exception is generated that results in the overwritten `Handler` being called. As a result of this, the attacker can more easily gain control of code execution due to the context that the exception handler is called in.

Microsoft has attempted to address the problem of SEH overwrites with enhancements to the exception dispatcher itself and with solutions like SafeSEH and the `/GS` compiler flag. However, these solutions are limited because they require a recompilation of code and therefore only protect images that have been compiled with these flags enabled. This limitation is something that Microsoft is aware of and it was most likely chosen to reduce the potential for compatibility issues.

To help solve the problem of not offering complete protection against SEH overwrites, this paper has suggested a solution that can be used without any code recompilation and with negligible performance overhead. The solution involves appending a custom exception registration record, known as a *validation frame*, to the end of the exception list early on in thread startup. When an exception occurs in the context of a thread, the solution intercepts the exception and validates the exception handler chain for the thread by making sure that it can walk the chain until it reaches the validation frame. If it is able to reach the validation frame, then the exception is dispatched like normal. However, if the validation frame cannot be reached, then it is assumed that the exception handler chain is corrupt and that it's possible that an exploit attempt may have

occurred. Since exception registration records are always prepended to the exception handler chain, the validation frame is guaranteed to always be the last handler.

This solution relies on the fact that when an SEH overwrite occurs, the `Next` attribute is overwritten before overwriting the `Handler` attribute. Due to the fact that attackers typically use the `Next` attribute as the location at which to store a short jump, it is not possible for them to both retain the integrity of the list and also use it as a location to store code. This important consequence is the key to being able to detect and prevent the leveraging of an SEH overwrite to gain code execution.

Looking toward the future, the usefulness of this solution will begin to wane as 64-bit versions of Windows begin to dominate the desktop environment. The reason 64-bit versions are not affected by this solution is because exception handling on 64-bit versions of Windows is inherently secure due to the way it's been implemented[8]. However, this only applies to 64-bit binaries. Legacy 32-bit binaries that are capable of running on 64-bit versions of Windows will continue to use the old style of exception handling, thus potentially leaving them vulnerable to the same style of attacks depending on what compiler flags were used. On the other hand, this solution will also become less necessary due to the fact that modern 32-bit x86 machines support hardware NX and can therefore help to mitigate the execution of code from the stack. Regardless of these facts, there will always be a legacy need to protect against SEH overwrites, and the solution described in this paper is one method of providing that protection.

Bibliography

[1] Borland. *United States Patent: 5628016.* http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PALL&p=1&u=%2Fnetahtml%2FPTO%2Fsrchnum.htm&r=1&f=G&l=50&s1=5,628,016.PN.&OS=PN/5,628,016&RS=PN/5,628,016; accessed Sep 5, 2006.

[2] Litchfield, David. *Defeating the Stack based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server.* http://www.blackhat.com/presentations/bh-asia-03/bh-asia-03-litchfield.pdf; accessed Sep 5, 2006.

[3] Microsoft Corporation. *Structured Exception Handling.* http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/structured_exception_handling.asp; accessed Sep 5, 2006.

[4] Microsoft Corporation. *Working with the AppInit_DLLs registry value.* http://support.microsoft.com/default.aspx?scid=kb;en-us;197571; accessed Sep 5, 2006.

[5] Microsoft Corporation. */GS (Buffer Security Check)* http://msdn2.microsoft.com/en-us/library/8dbf701c.aspx; accessed Sep 5, 2006.

[6] Nagy, Ben. *SEH (Structured Exception Handling) Security Changes in XPSP2 and 2003 SP1.* http://www.eeye.com/html/resources/newsletters/vice/VI20060830.html#vexposed; accessed Sep 8, 2006.

[7] Pietrek, Matt. *A Crash Course on the Depths of Win32 Structured Exception Handling.* http://www.microsoft.com/msj/0197/exception/exception.aspx; accessed Sep 8, 2006.

[8] skape. *Improving Automated Analysis of Windows x64 Binaries.* http://www.uninformed.org/?v=4&a=1&t=sumry; accessed Sep 5, 2006.

[9] Wehnus. *WehnTrust*.
http://www.wehnus.com/products.pl; accessed Sep 5, 2006.

[10] Wikipedia. *Matryoshka Doll*.
http://en.wikipedia.org/wiki/Matryoshka_doll; accessed Sep 18, 2006.

[11] Wine. *CompilerExceptionSupport*.
http://wiki.winehq.org/CompilerExceptionSupport; accessed Sep 5, 2006.