# ECE 6504

Embedded and Controls Systems Security: Stack Smashing IV





"I intend to live forever, or die trying."

-Groucho Marx



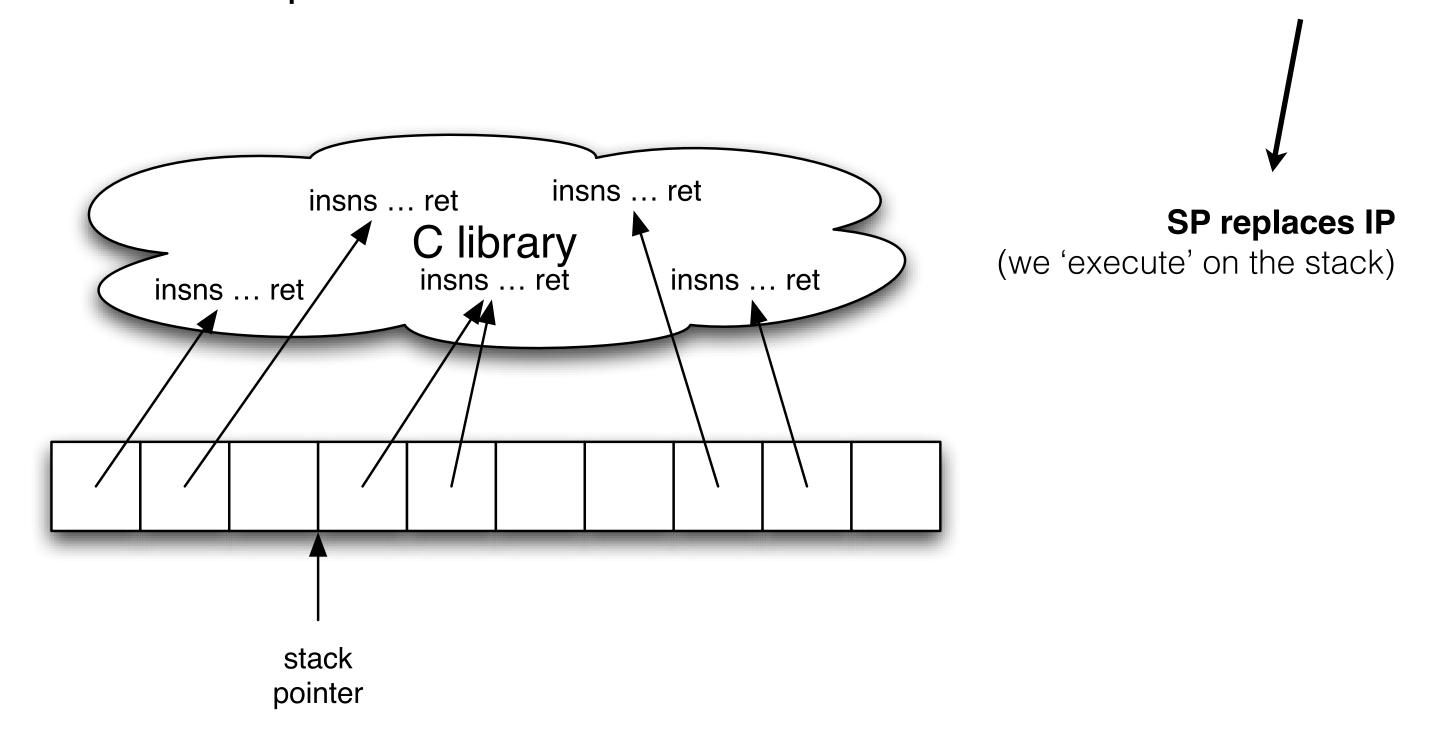


setup stack to point to instructions that end in return

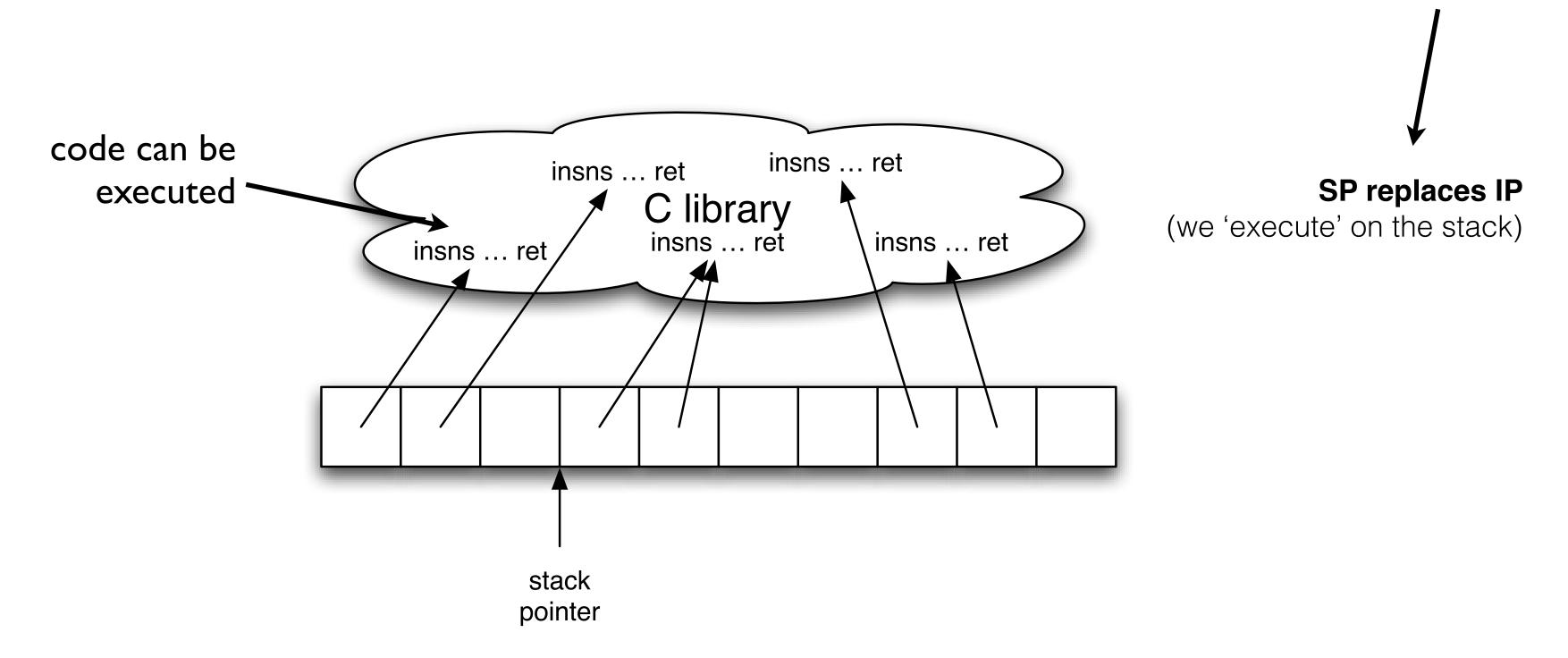


(we 'execute' on the stack)

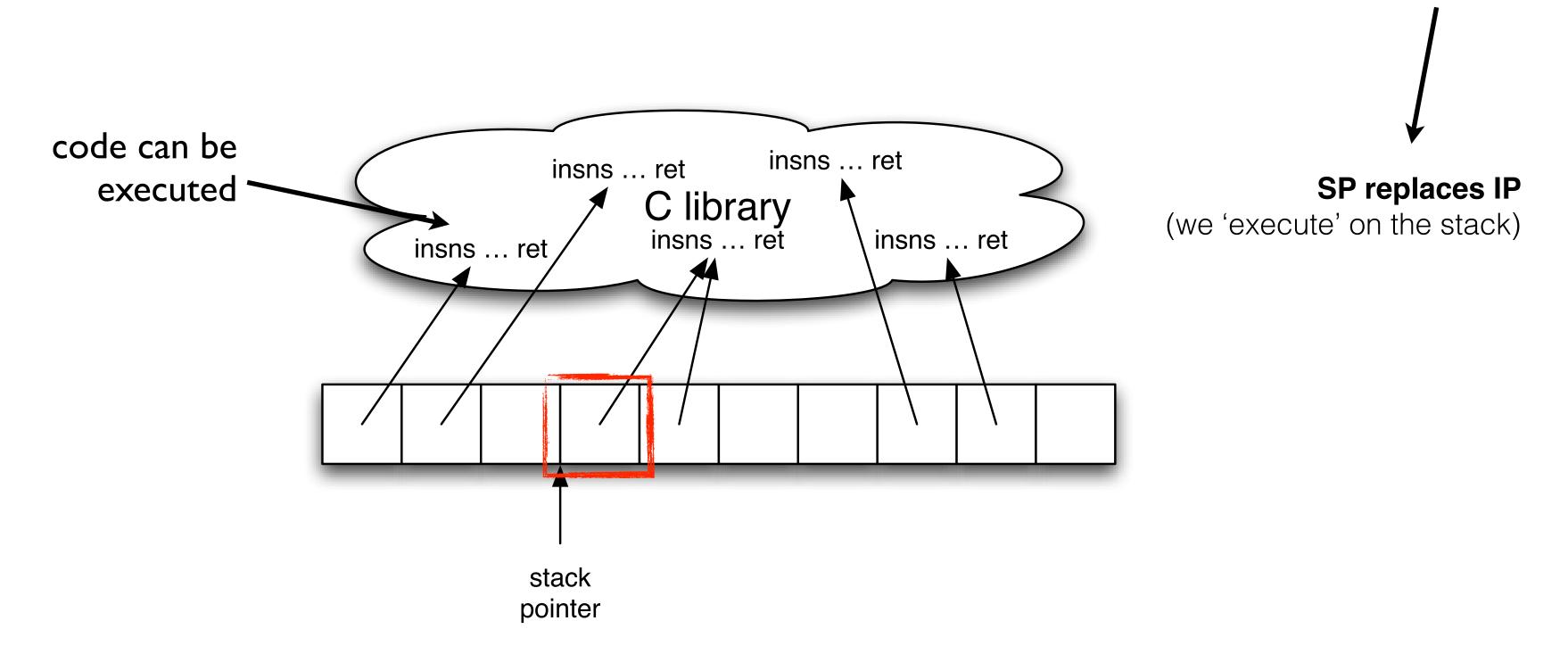




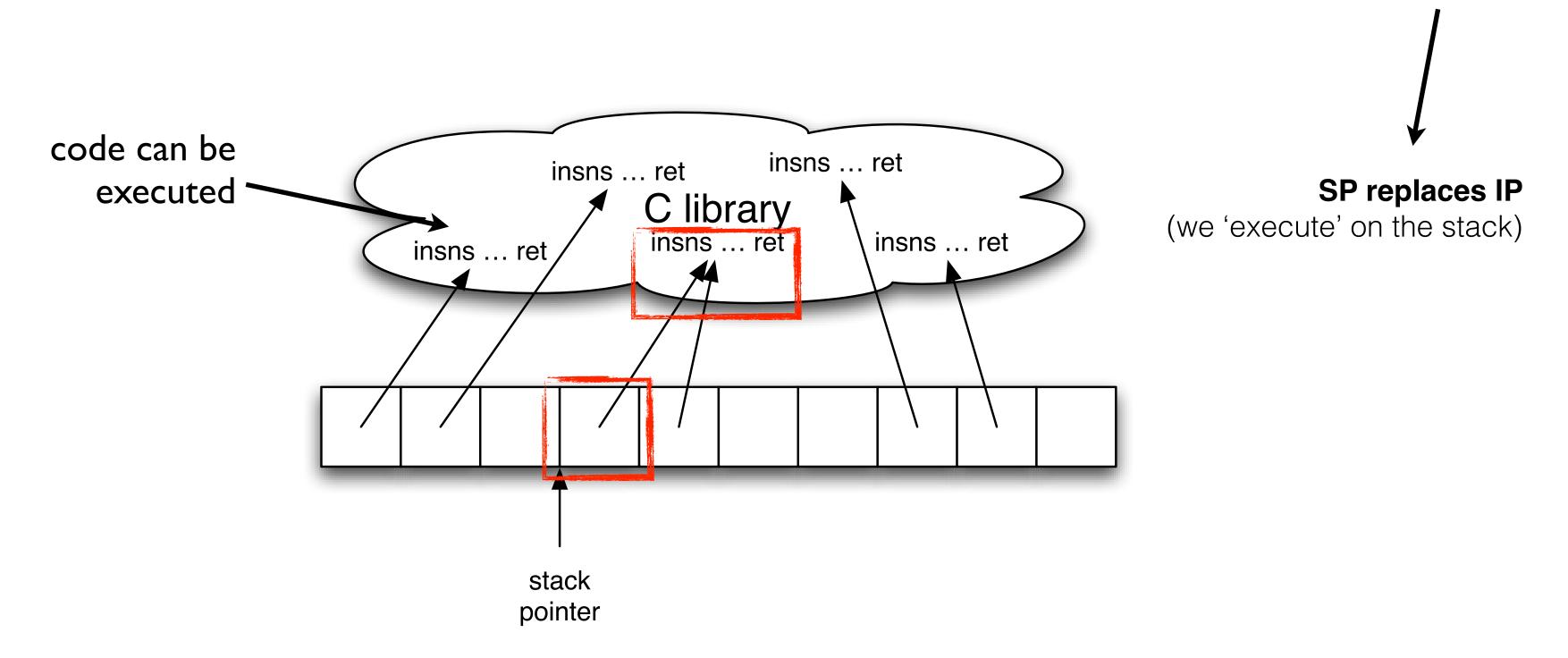






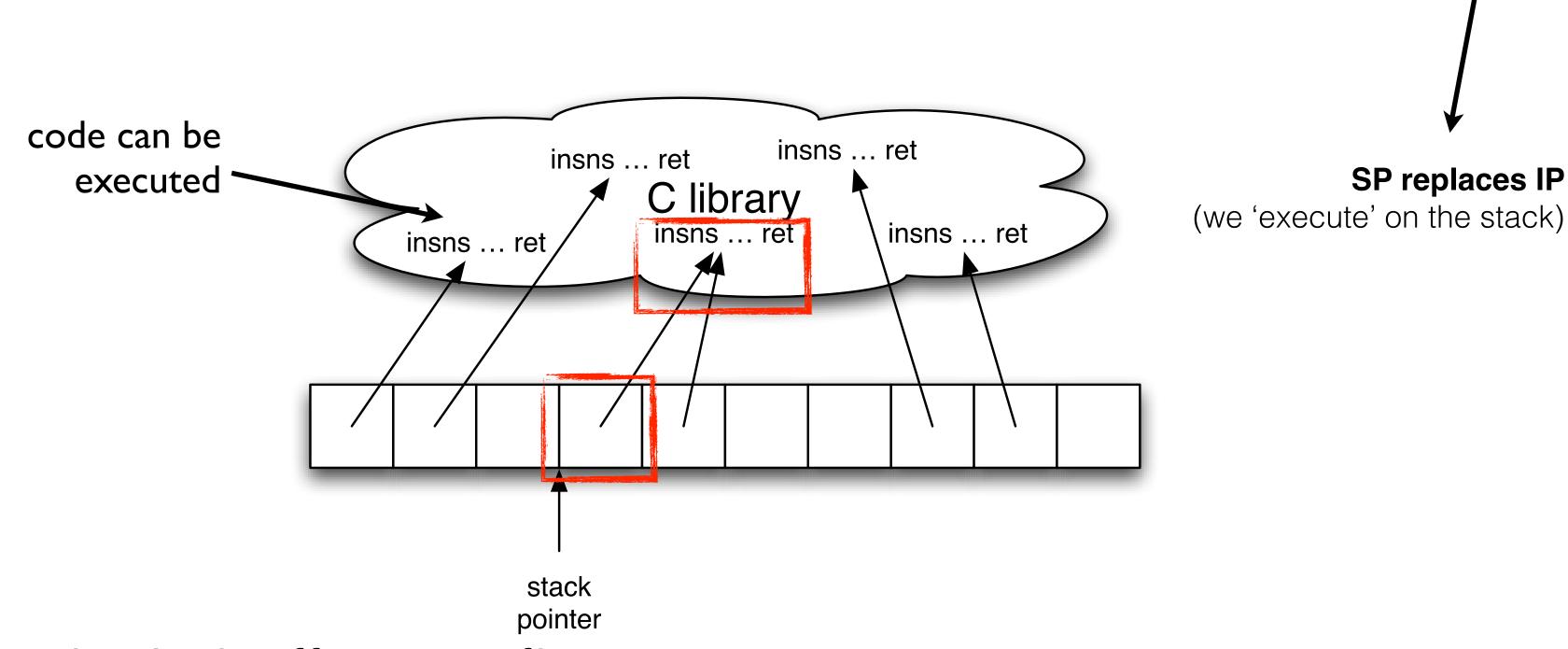








setup stack to point to instructions that end in return



procedure:

1. setup stack via buffer overflow

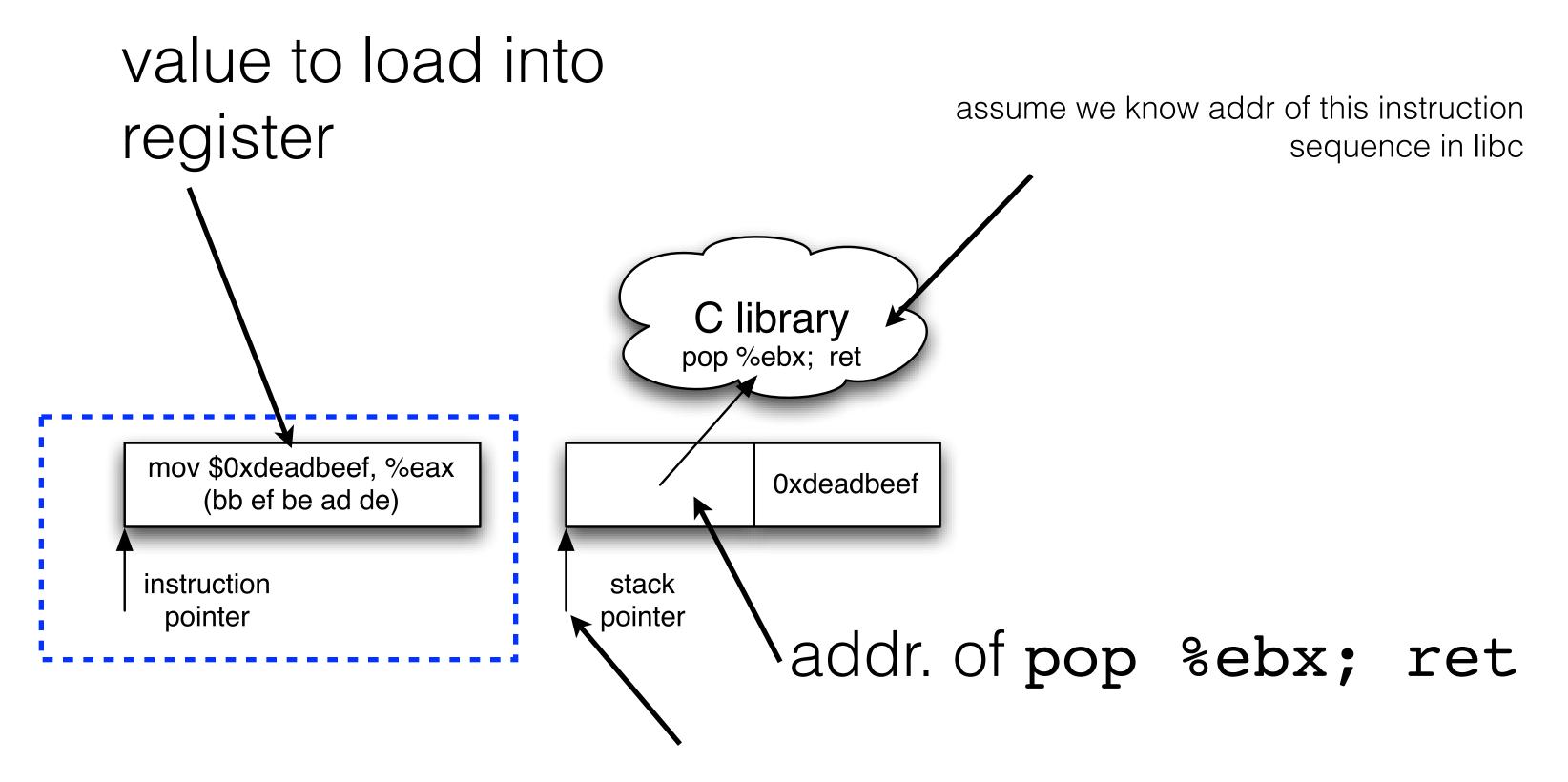
2. issue return

3. pop addr from stack

4. IP executes instruction at addr



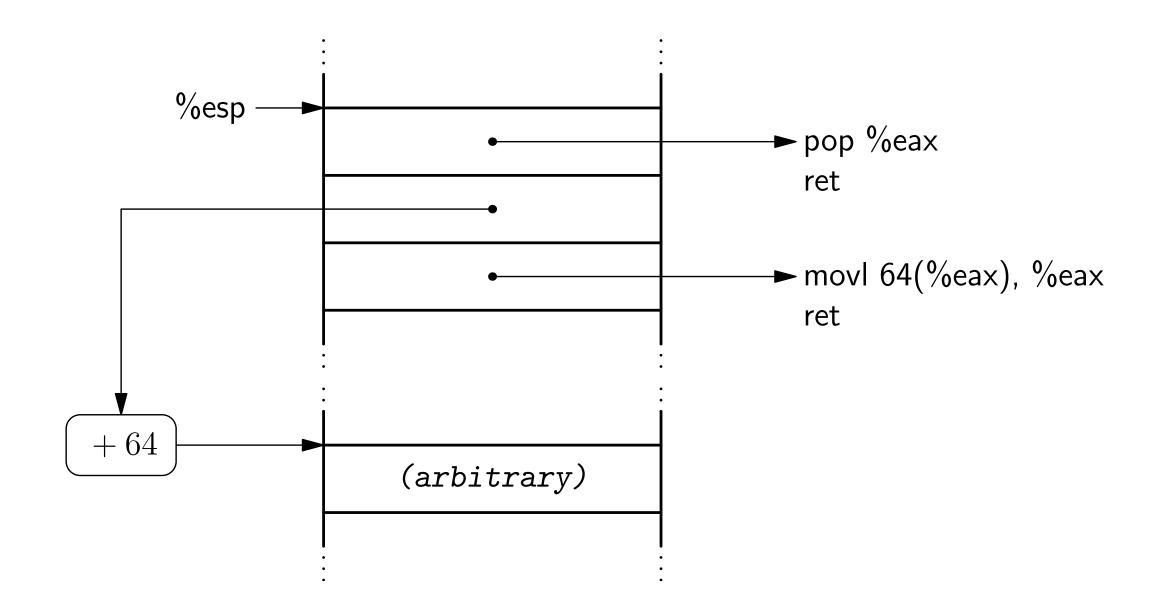
#### ROP immediate load

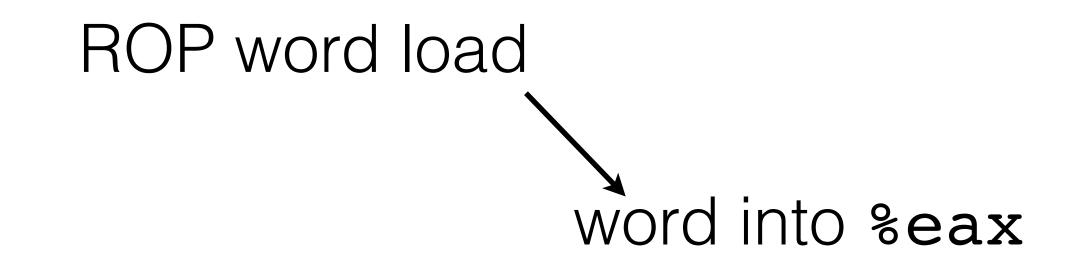


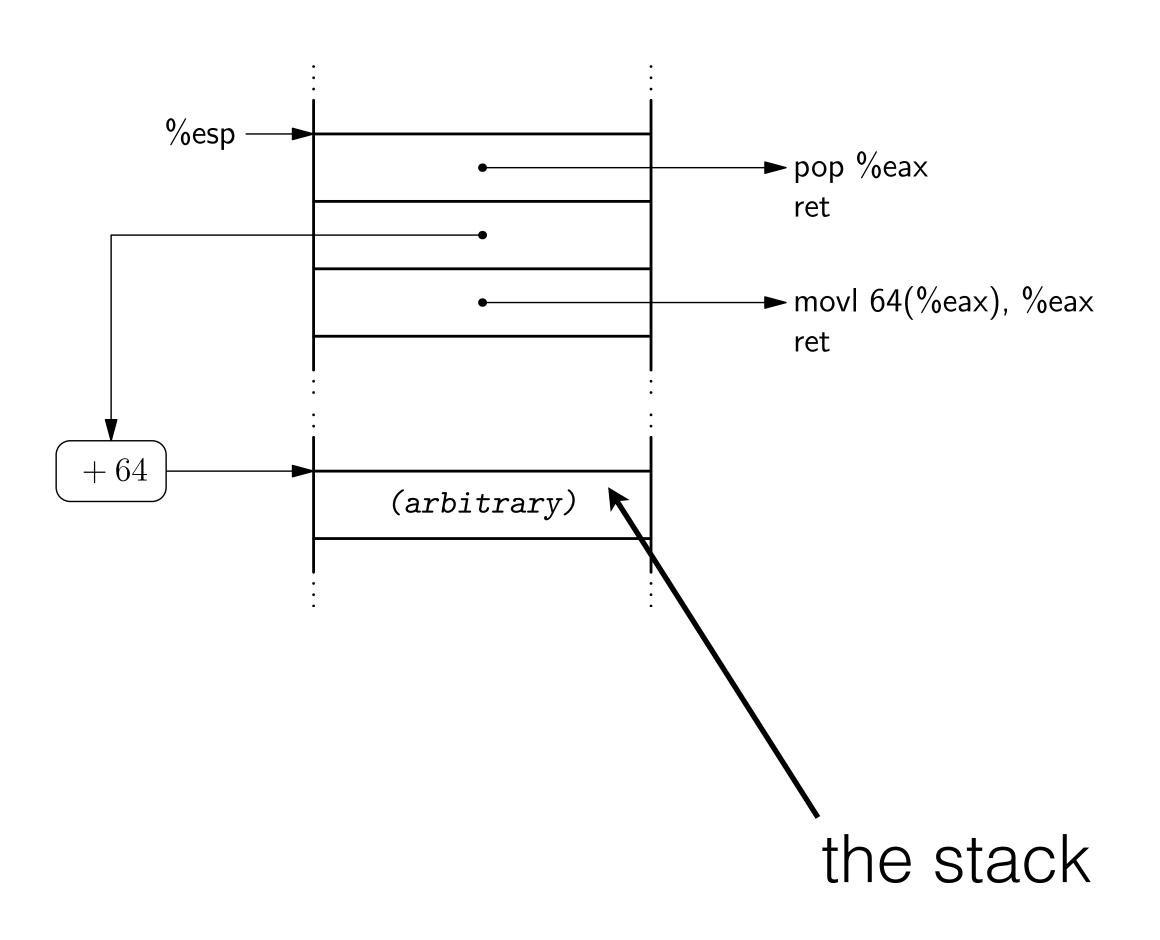
- 1. ret issued with SP here
- 2. IP pointed at addr of pop...
  - 3. SP moved right one word
- 4. pop puts 0xdeadbeef into ebx
  - 5. SP moved right one word



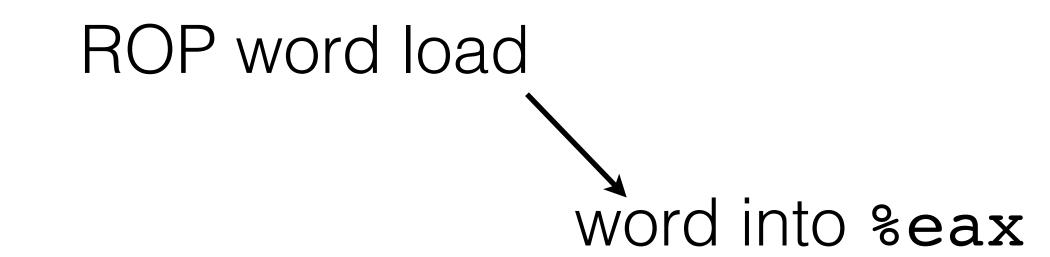
#### ROP word load



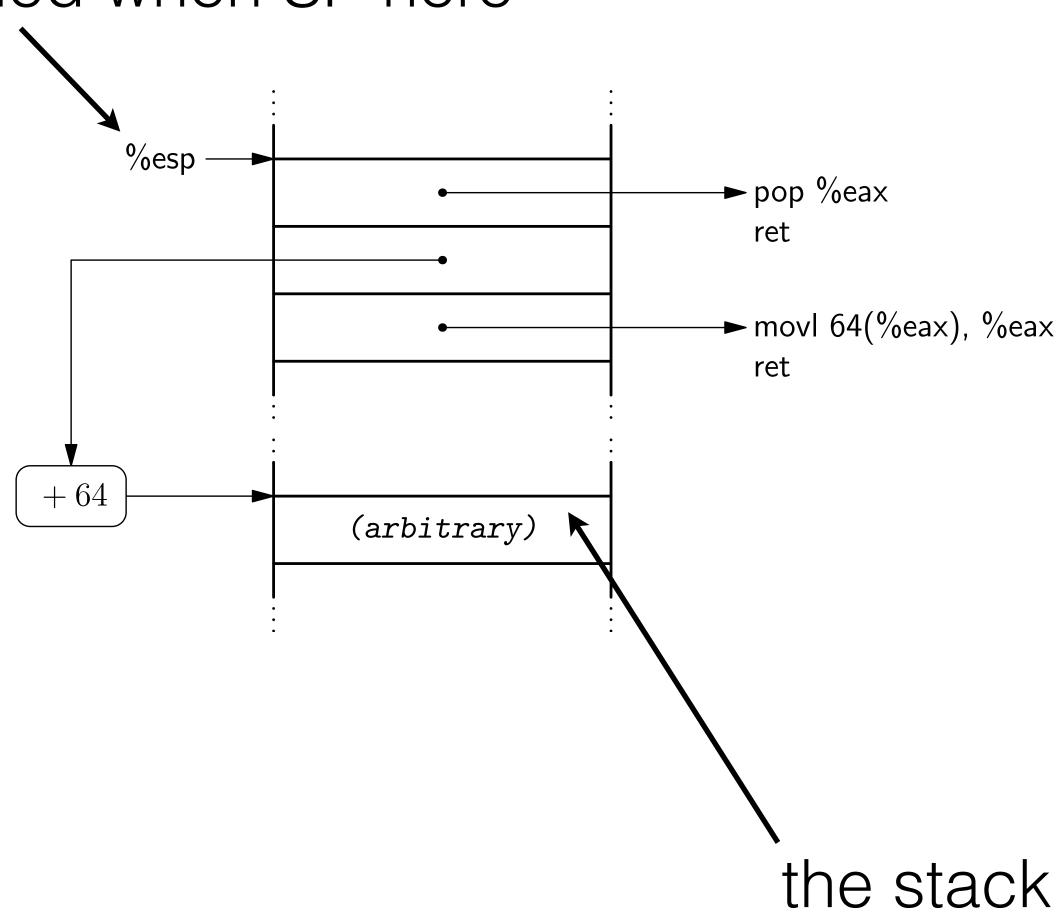




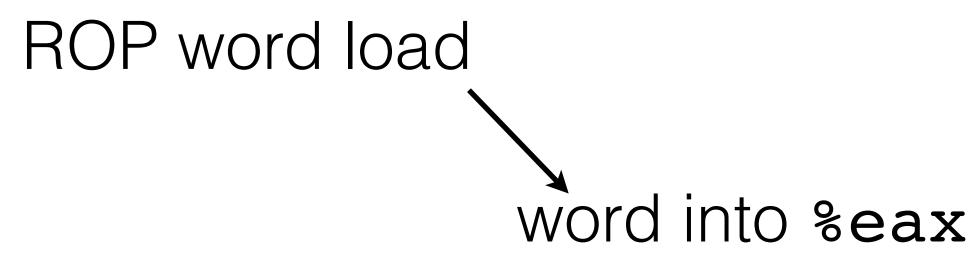


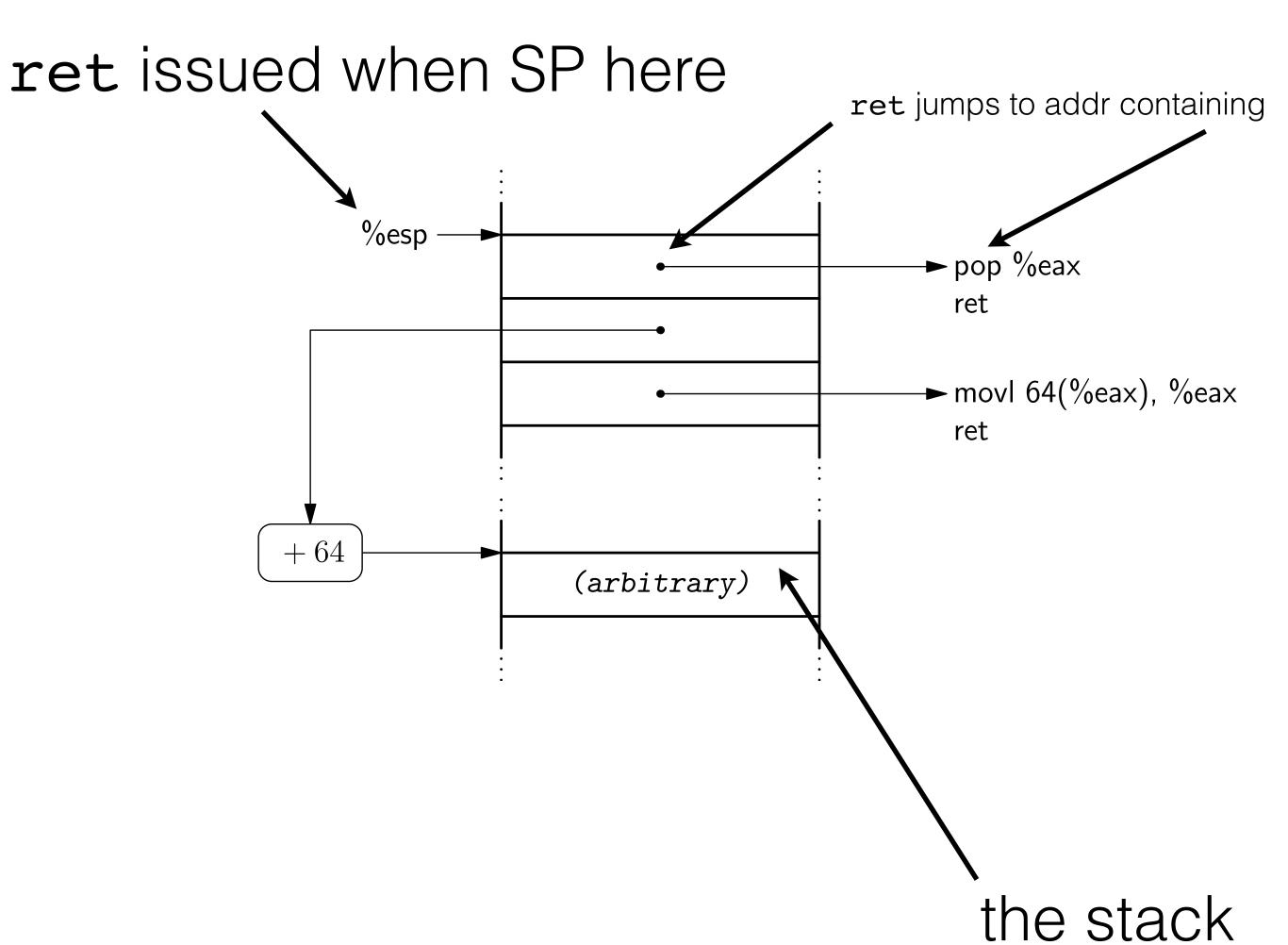


#### ret issued when SP here

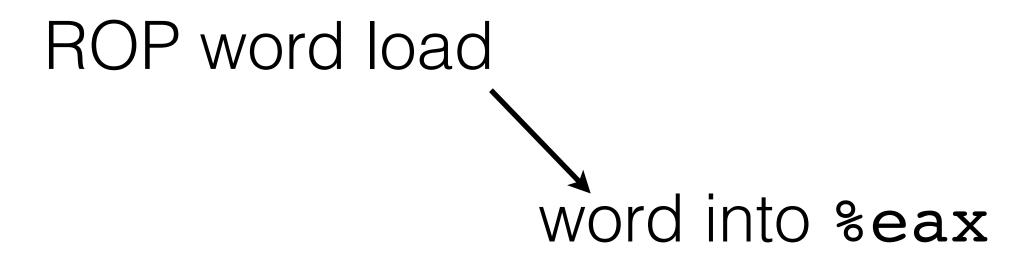


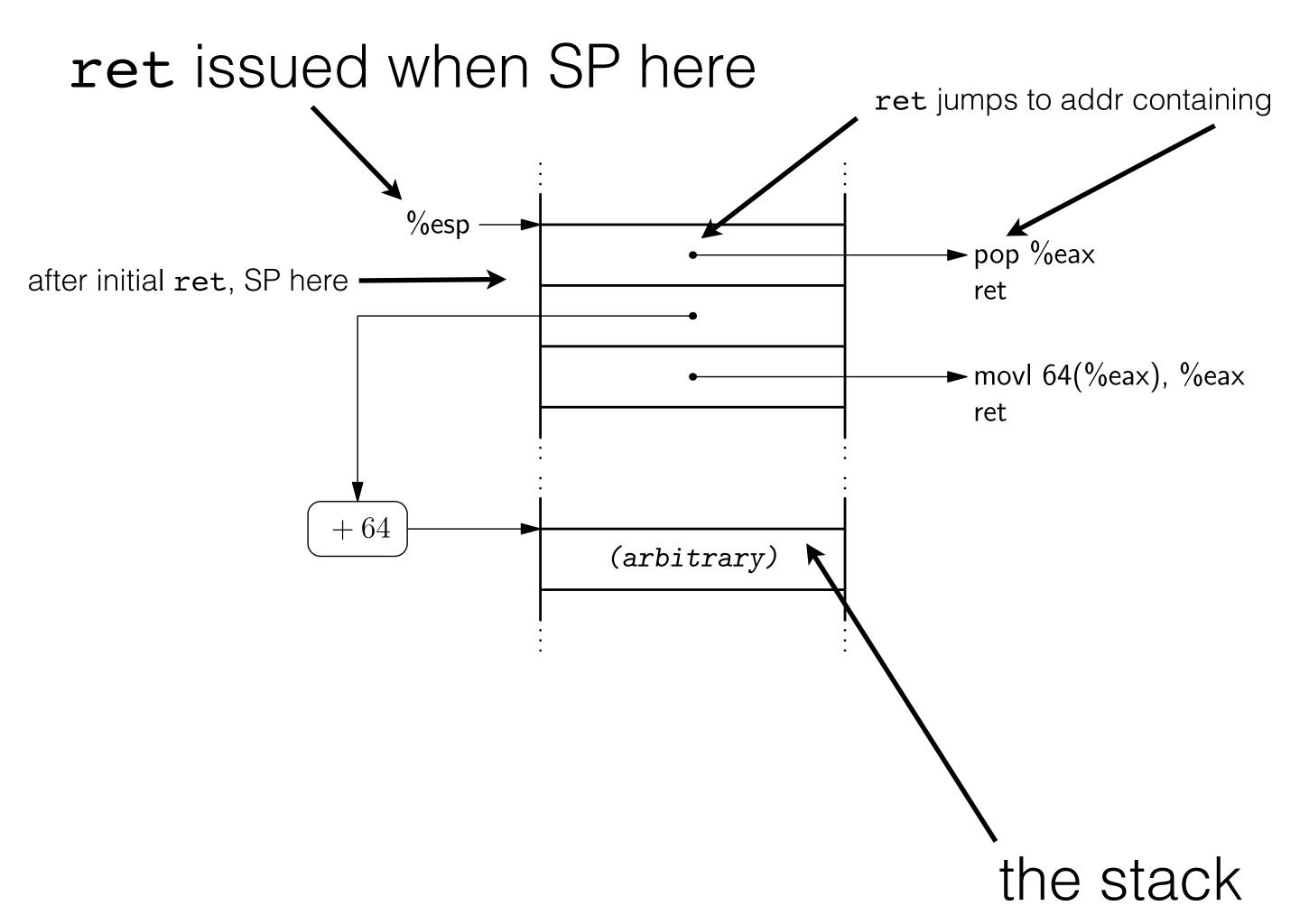




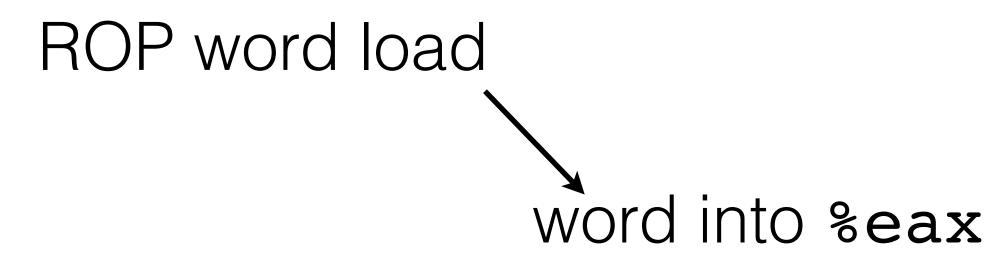


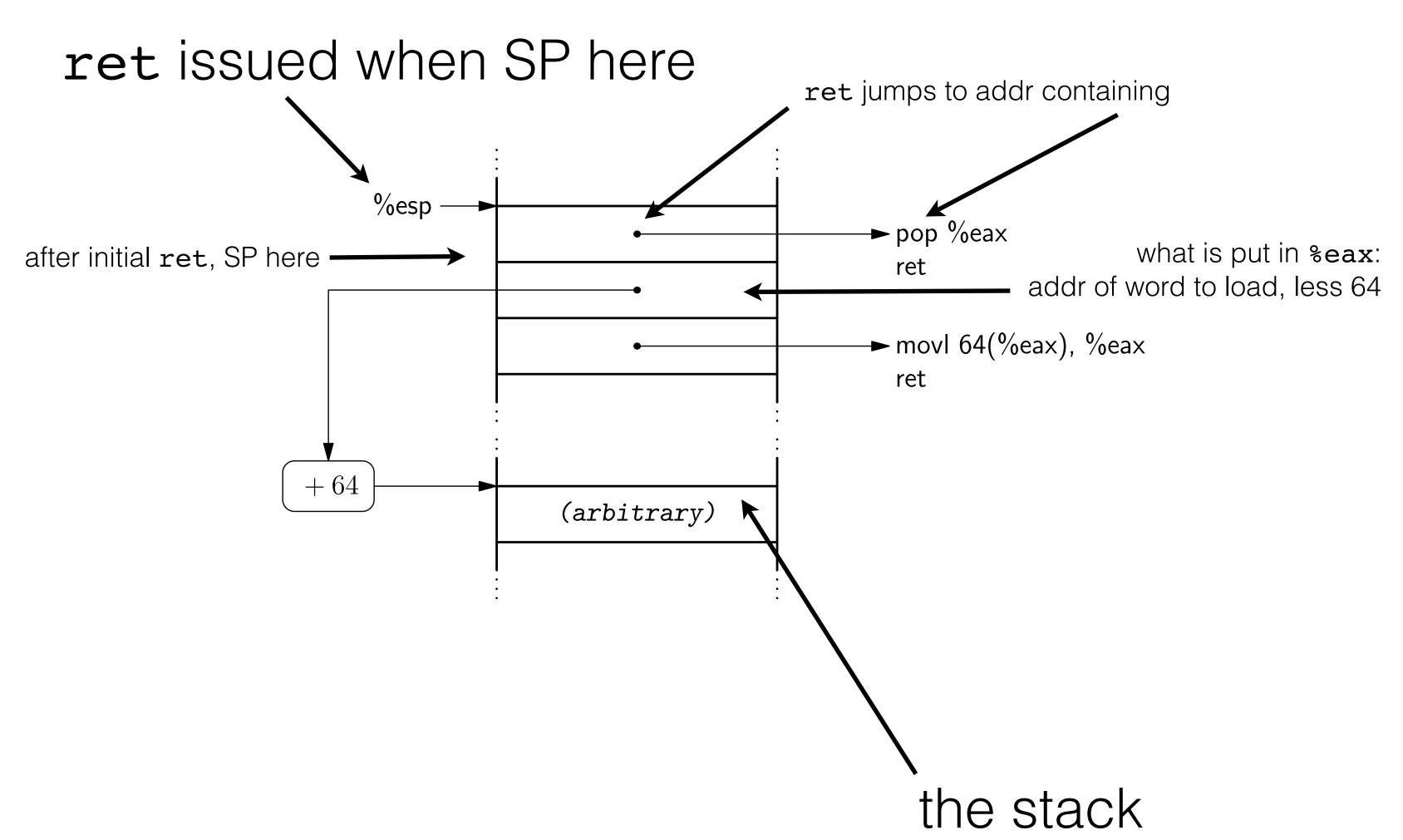


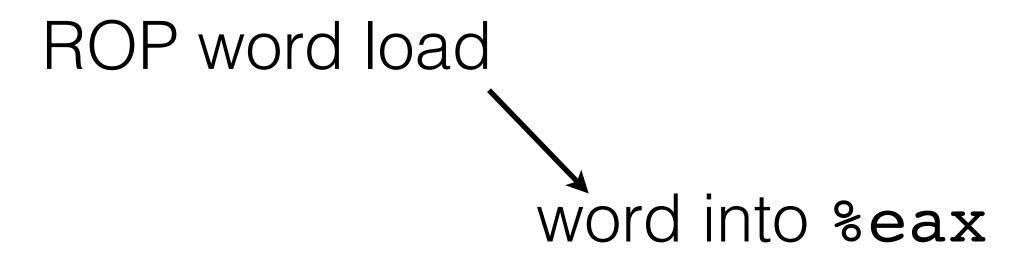


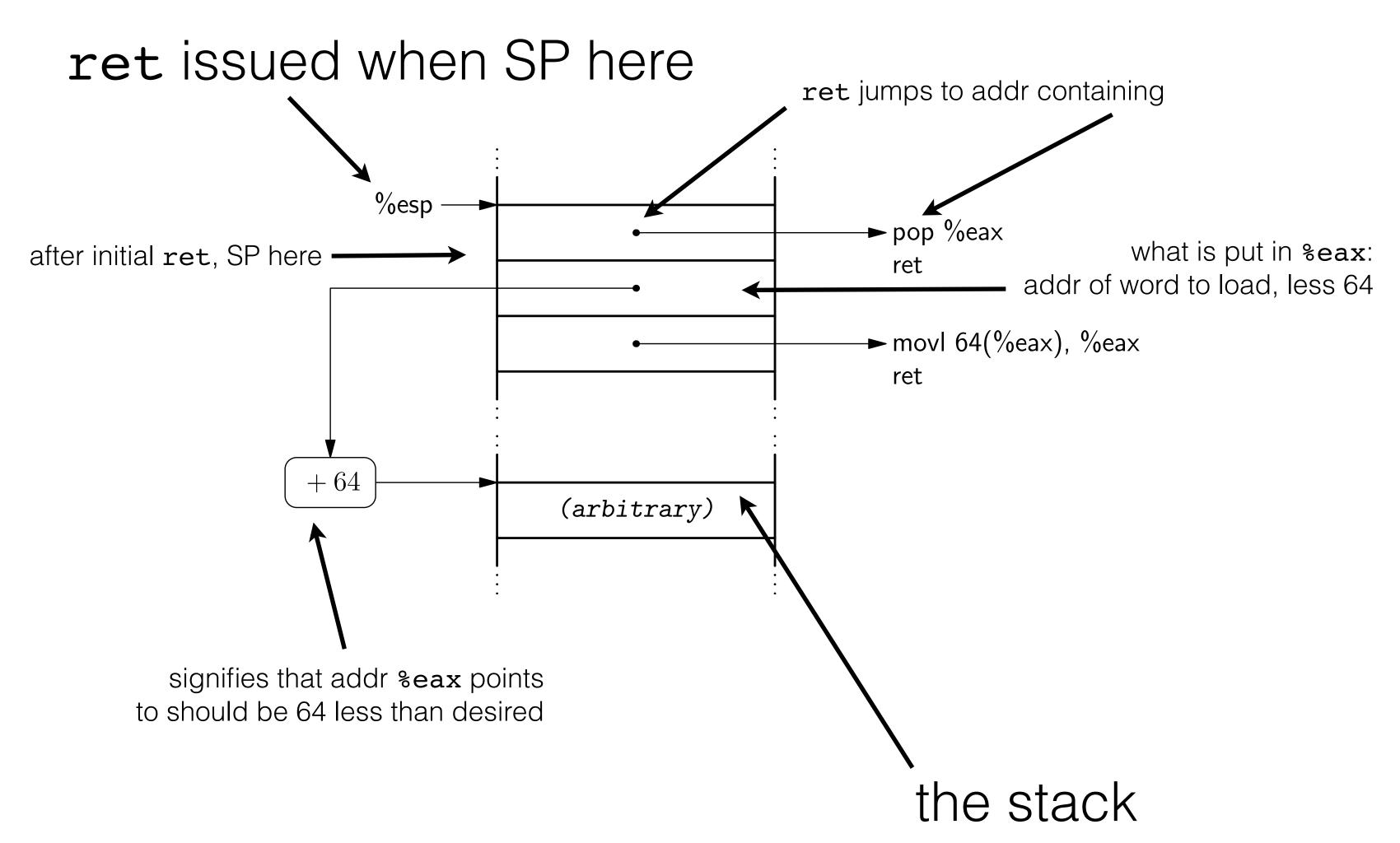


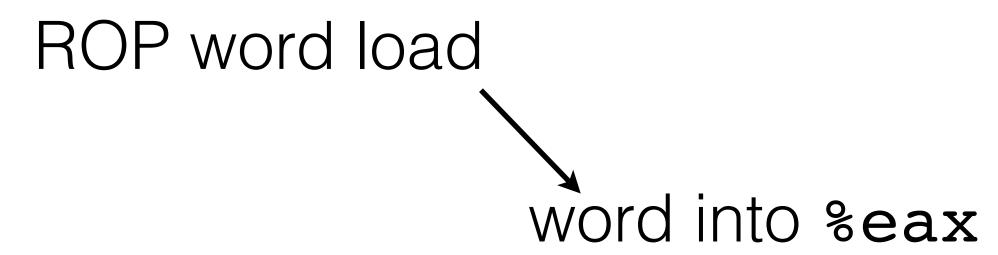


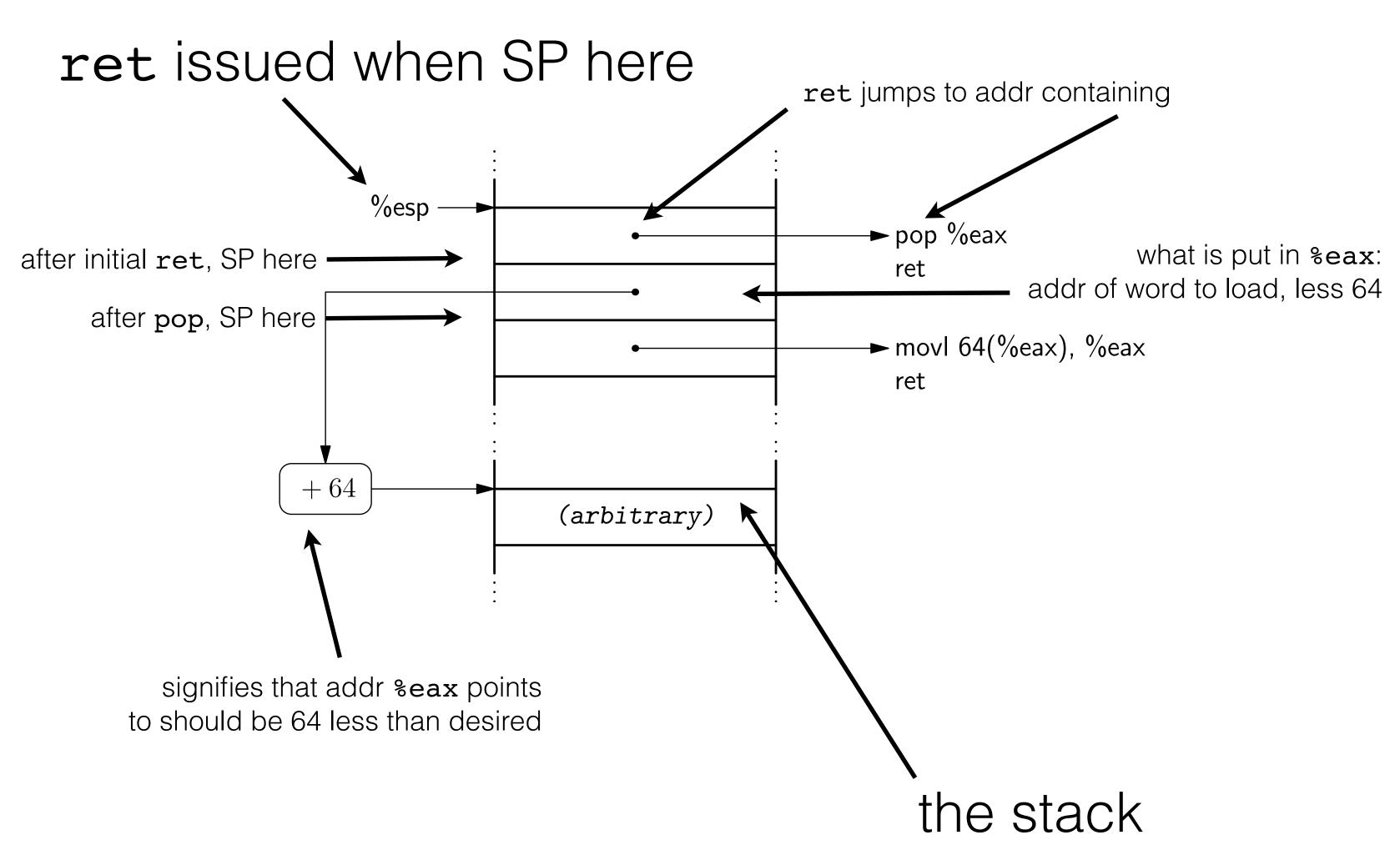


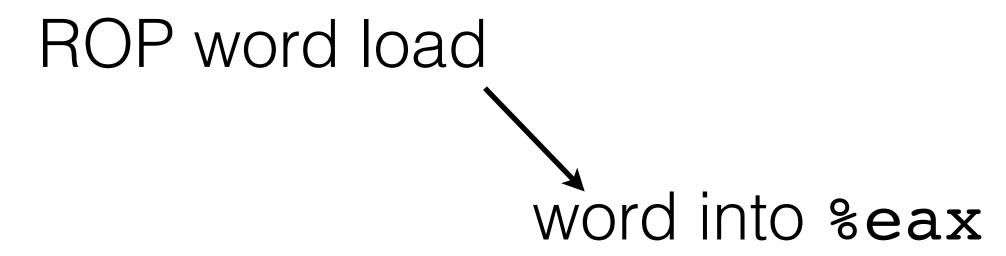


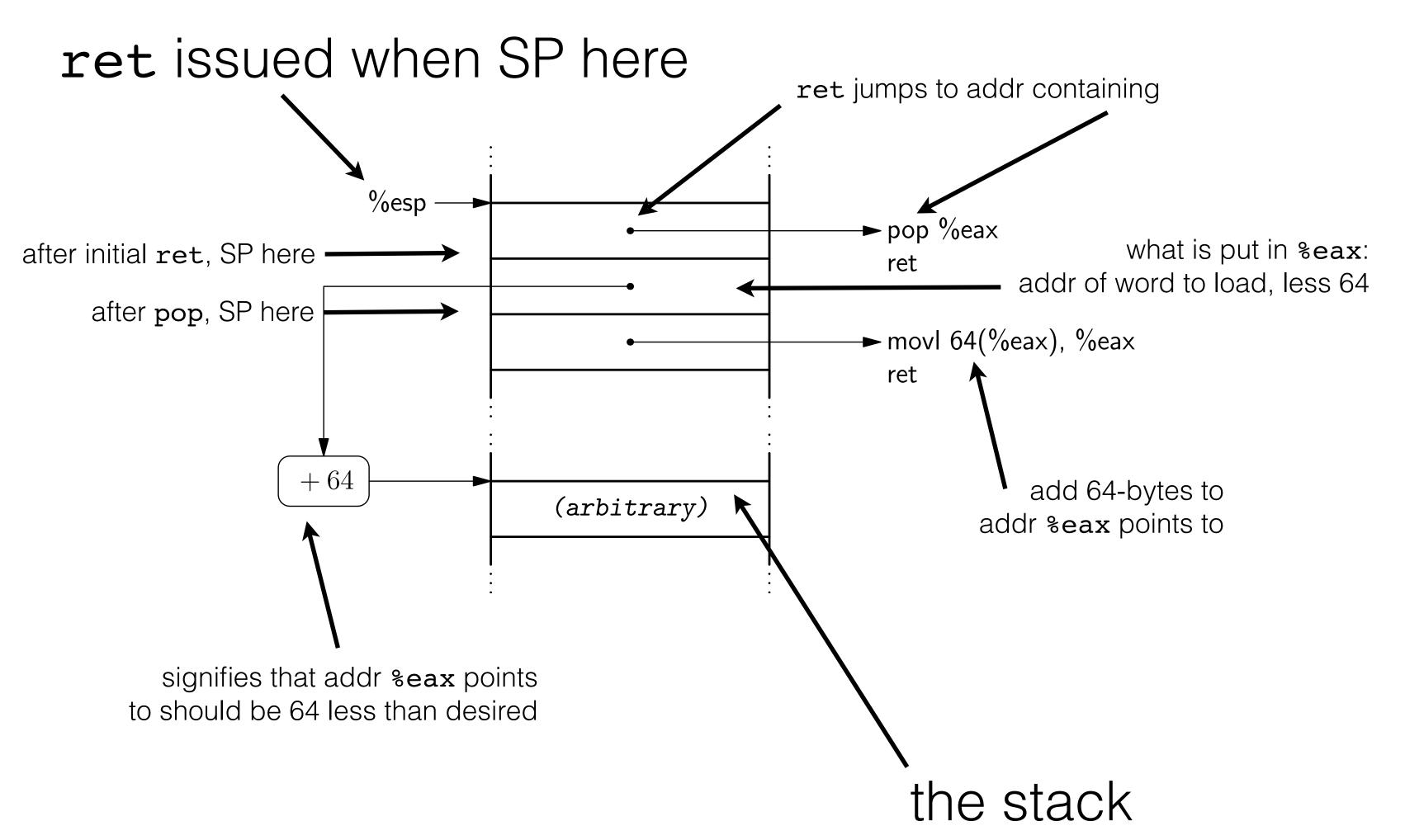


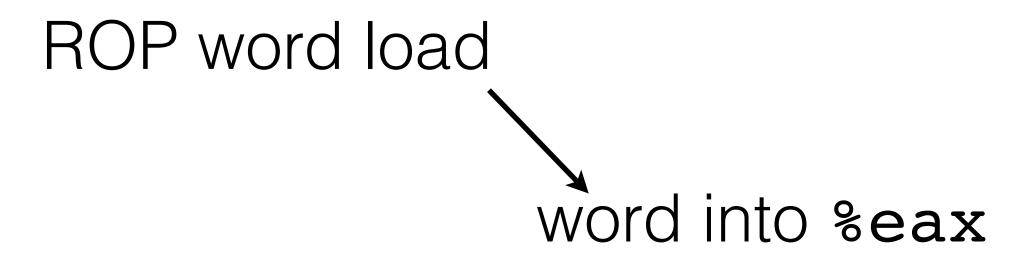


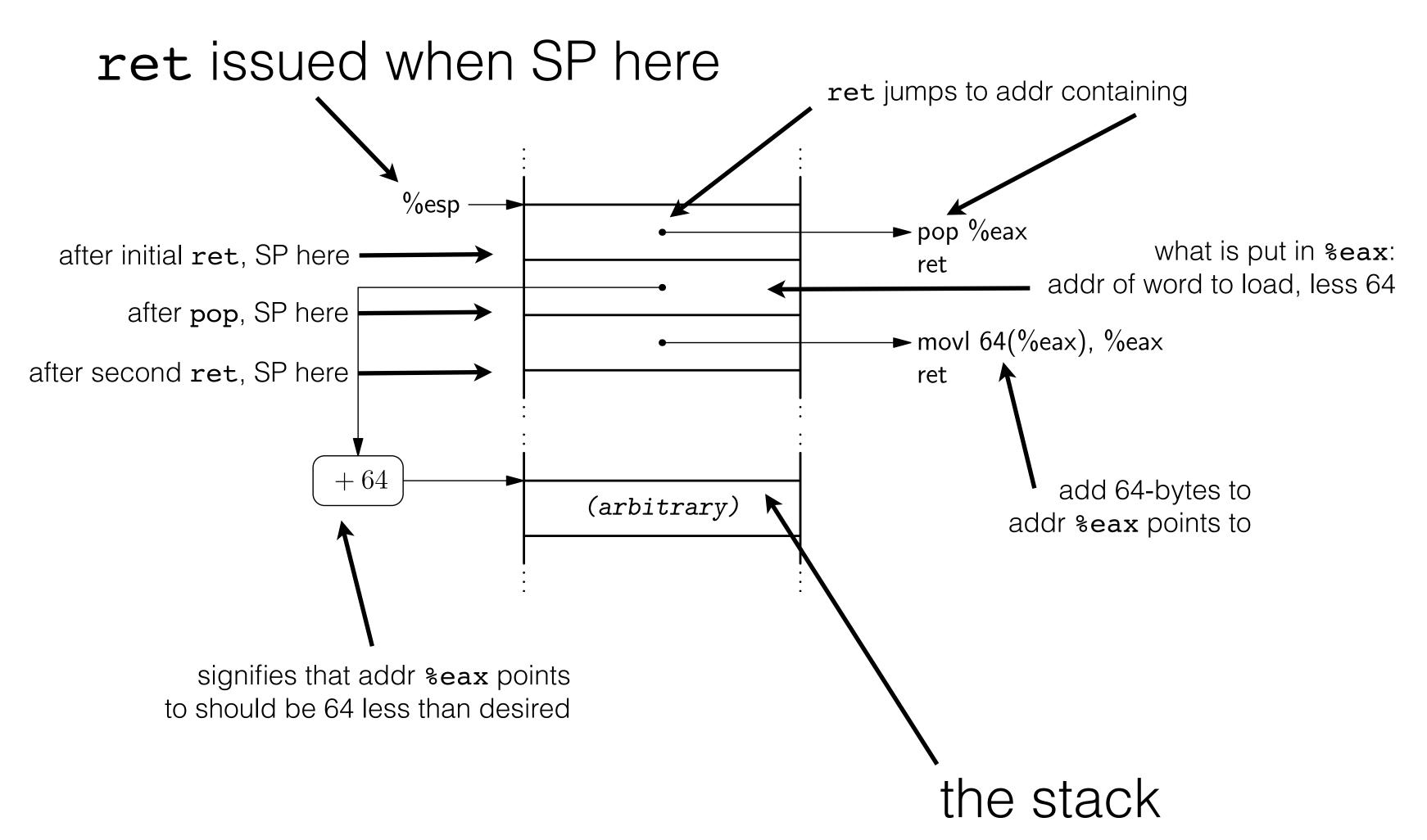














can sequence gadgets s.t. Turing complete instruction set...



Q: is this necessary for exploitation in all instances?

A: no, let's build gadgets for what we need (shellcode)



can sequence gadgets s.t. Turing complete instruction set...

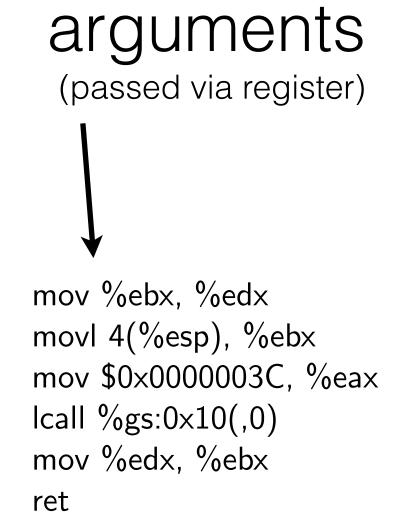


Q: is this necessary for exploitation in all instances?

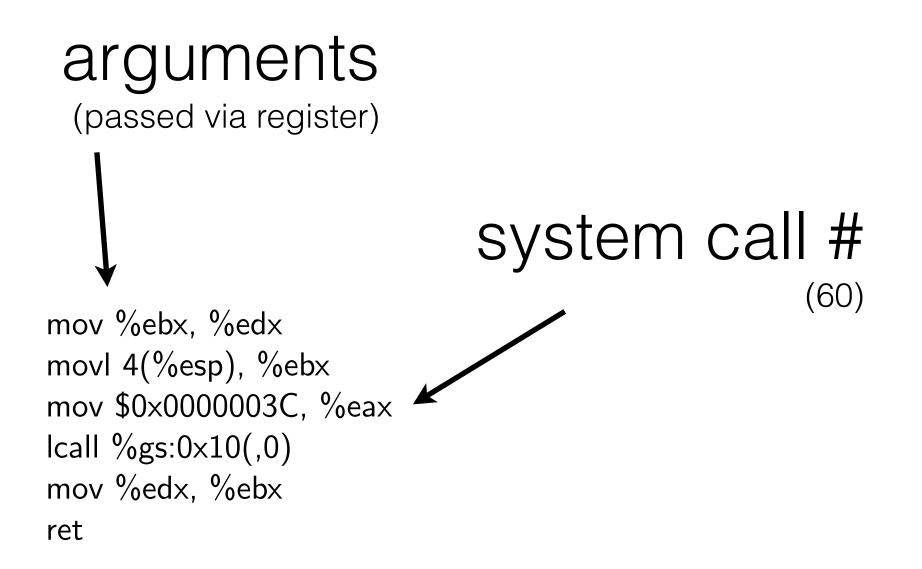
A: no, let's build gadgets for what we need (shellcode)



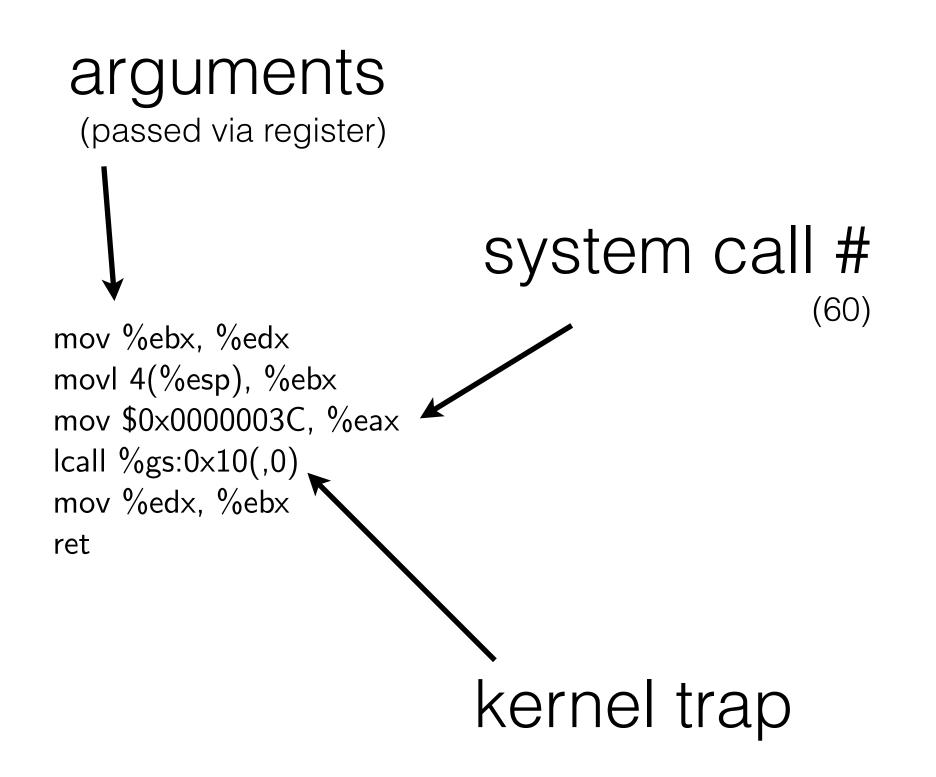




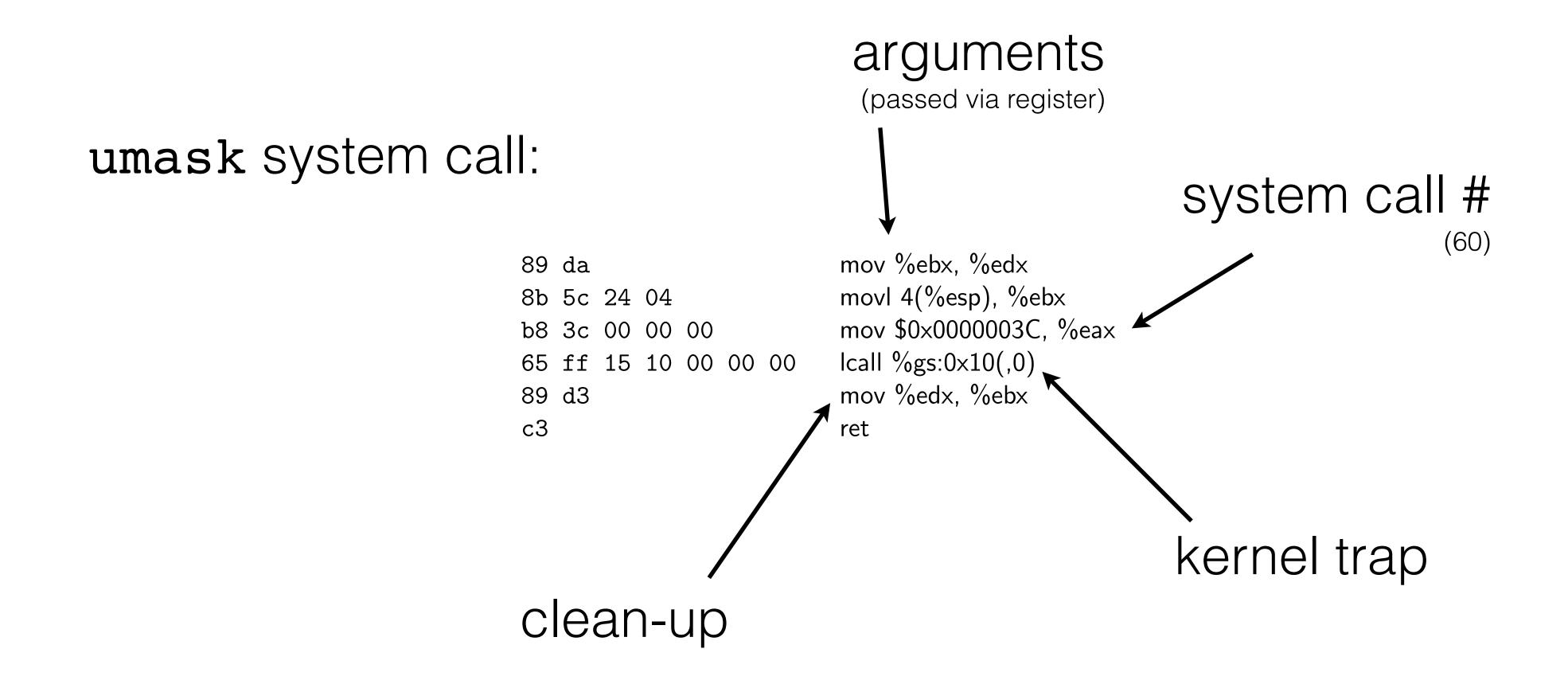














```
system call number 11 execve("/bin/sh", {"/bin/sh", NULL}, NULL);
```



```
system call number 11 execve("/bin/sh", {"/bin/sh", NULL}, NULL);
```

```
NOTE: int execve(const char *path, char *const argv[], char *const envp[])
```



```
system call number 11 execve("/bin/sh", {"/bin/sh", NULL}, NULL);
```

```
NOTE:int execve(const char *path, char *const argv[], char *const envp[])

path to program
```



```
system call number 11 execve("/bin/sh", {"/bin/sh", NULL}, NULL);
```

```
NOTE: int execve(const char *path, char *const argv[], char *const envp[])

path to program

args for program
```



```
system call number 11 execve("/bin/sh", {"/bin/sh", NULL}, NULL);
```

```
NOTE: int execve(const char *path, char *const argv[], char *const envp[])

path to program

env vars. for program
```



```
system call number 11 execve("/bin/sh", {"/bin/sh", NULL}, NULL);
```

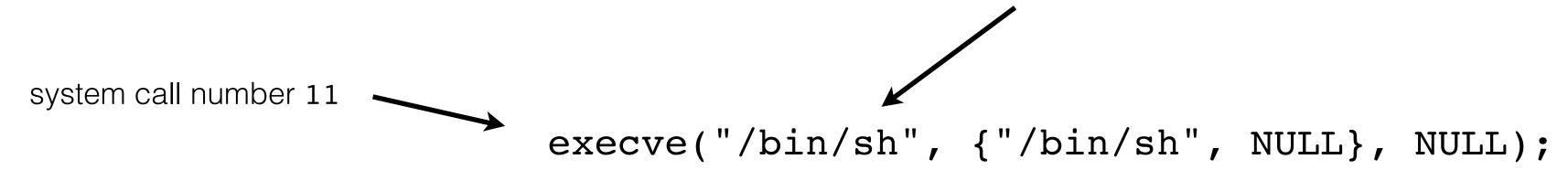
```
NOTE: int execve(const char *path, char *const argv[], char *const envp[])

path to program

env vars. for program
```

procedure:





```
NOTE: int execve(const char *path, char *const argv[], char *const envp[])

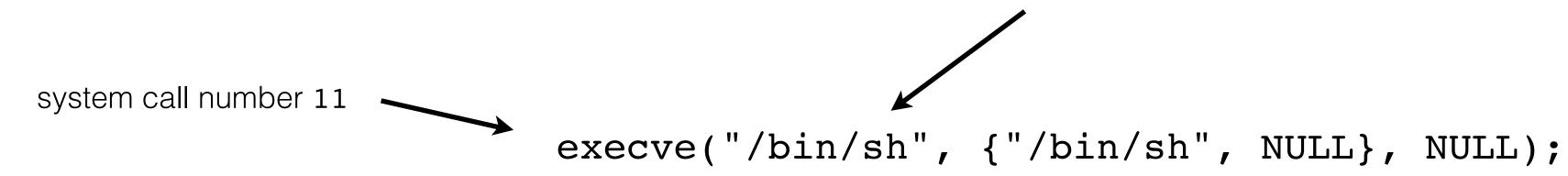
path to program

env vars. for program
```

procedure:

1. call # (0xb) into %eax





```
NOTE:int execve(const char *path, char *const argv[], char *const envp[])

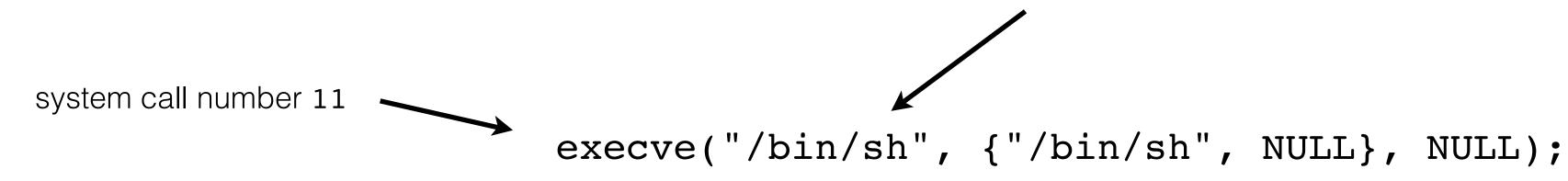
path to program

env vars. for program
```

#### procedure:

1. call # (0xb) into %eax 2. (ptr. to) program to run ('/bin/sh') in %ebx





```
NOTE: int execve(const char *path, char *const argv[], char *const envp[])

path to program

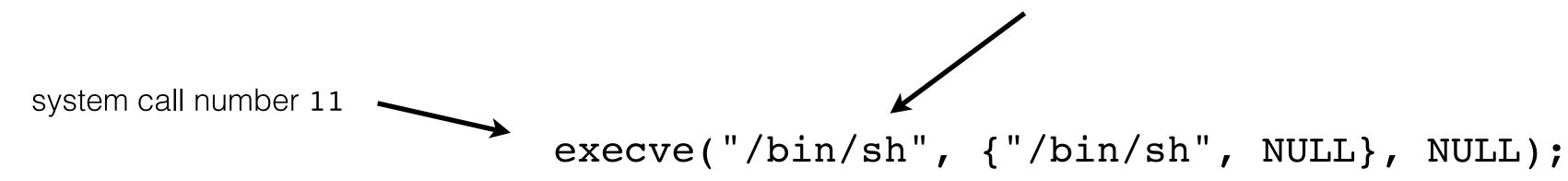
env vars. for program
```

## procedure:

- 1. call # (0xb) into %eax
- 2. (ptr. to) program to run ('/bin/sh') in %ebx
- 3. set argv: pointer to ptr. array ('/bin/sh' and NULL) in %ecx



## constructing ROP shellcode



```
NOTE:int execve(const char *path, char *const argv[], char *const envp[])

path to program

env vars. for program
```

#### procedure:

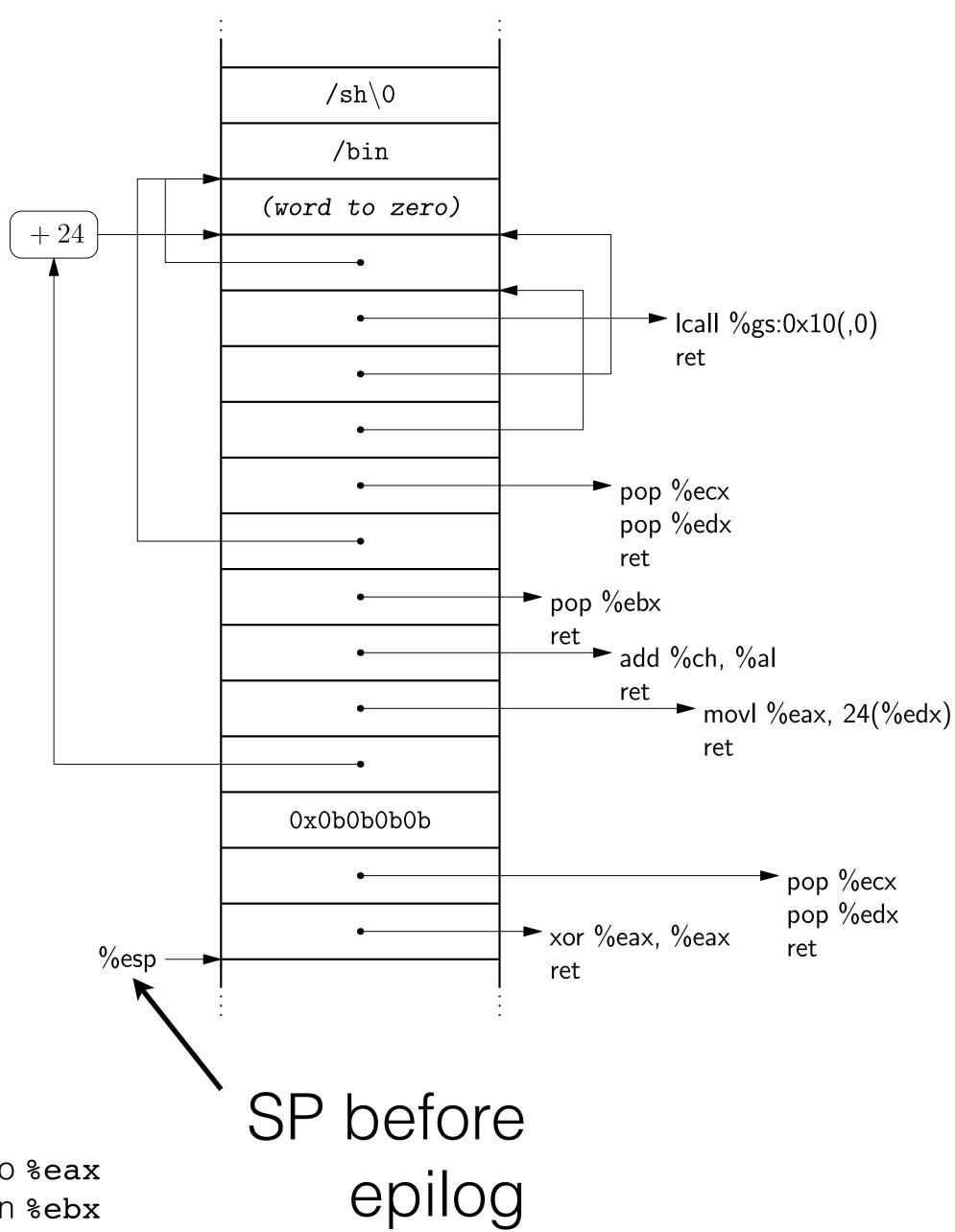
1. call # (0xb) into %eax

in %ecx

- 2. (ptr. to) program to run ('/bin/sh') in %ebx
- 3. set argv: pointer to ptr. array ('/bin/sh' and NULL)
  - 4. envp: pointer to NULL in %edx



# stack via buffer overflow



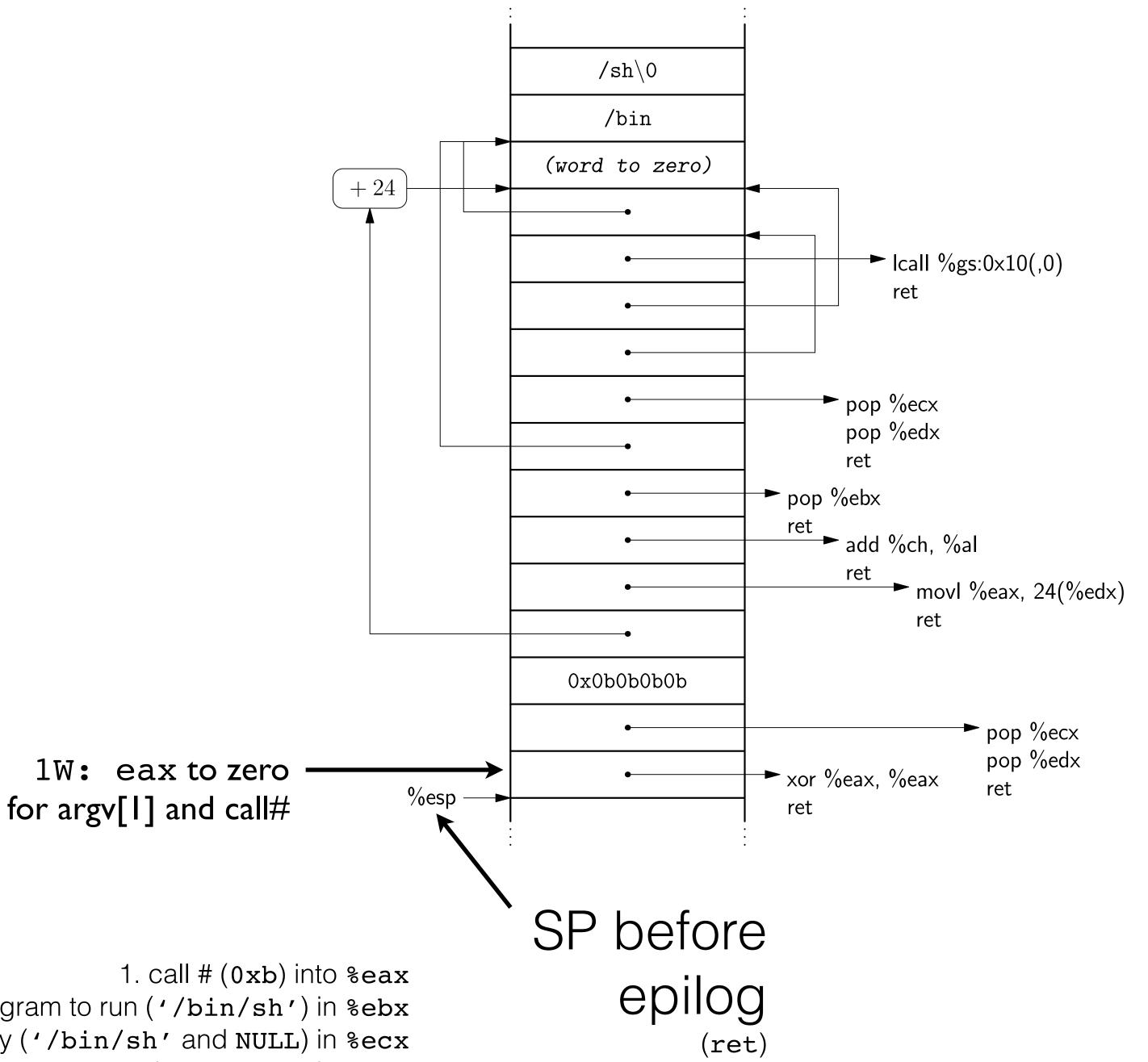
(ret)

procedure:

1. call # (0xb) into %eax
2. (ptr. to) program to run ('/bin/sh') in %ebx
3. set argv: pointer to ptr. array ('/bin/sh' and NULL) in %ecx
4. envp: pointer to NULL in %edx

Virginia Tech
Invent the Future

# stack via buffer overflow

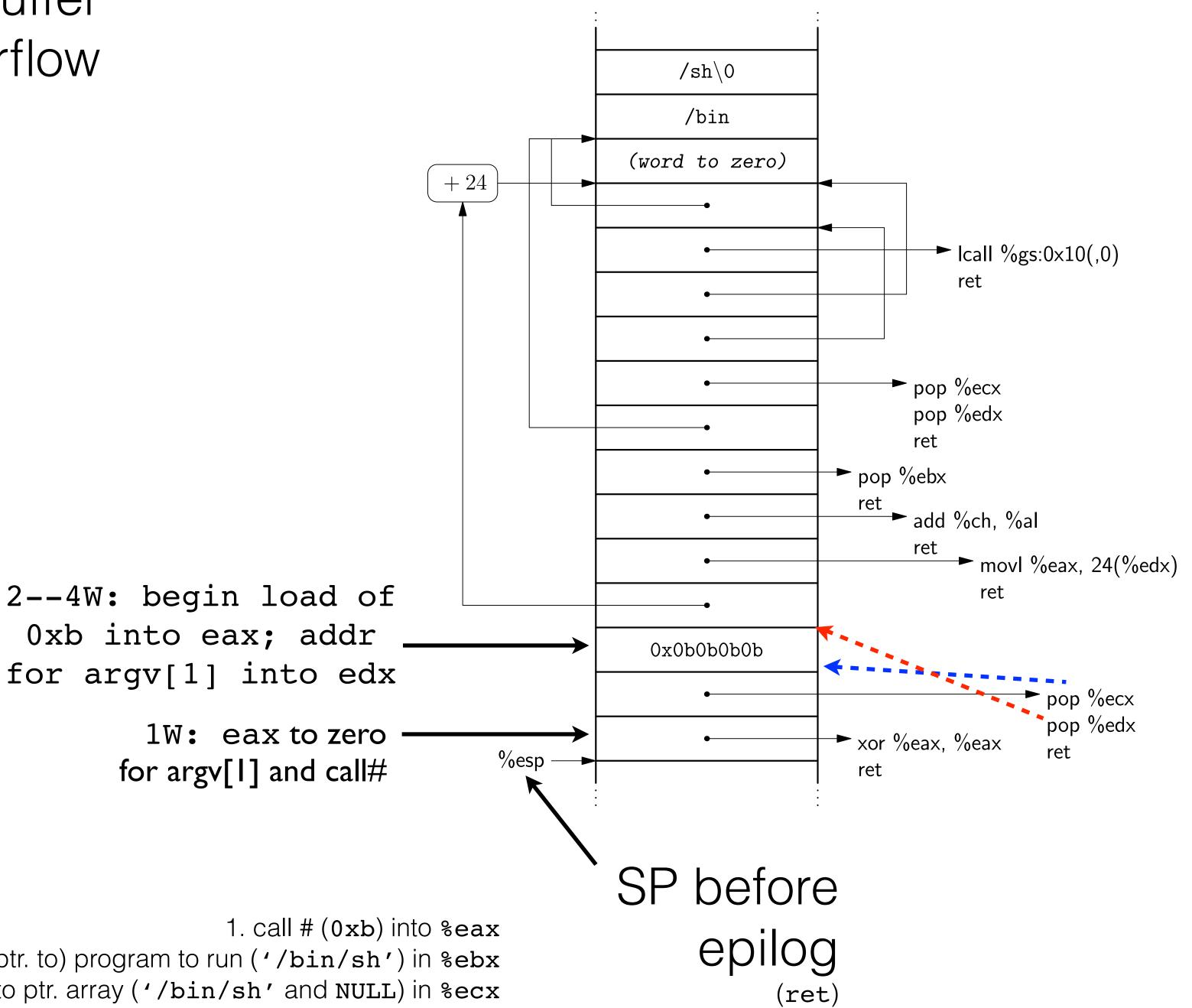


procedure:

1. call # (0xb) into %eax 2. (ptr. to) program to run ('/bin/sh') in %ebx 3. set argv: pointer to ptr. array ('/bin/sh' and NULL) in %ecx 4. envp: pointer to NULL in %edx

Virginia Tech
Invent the Future

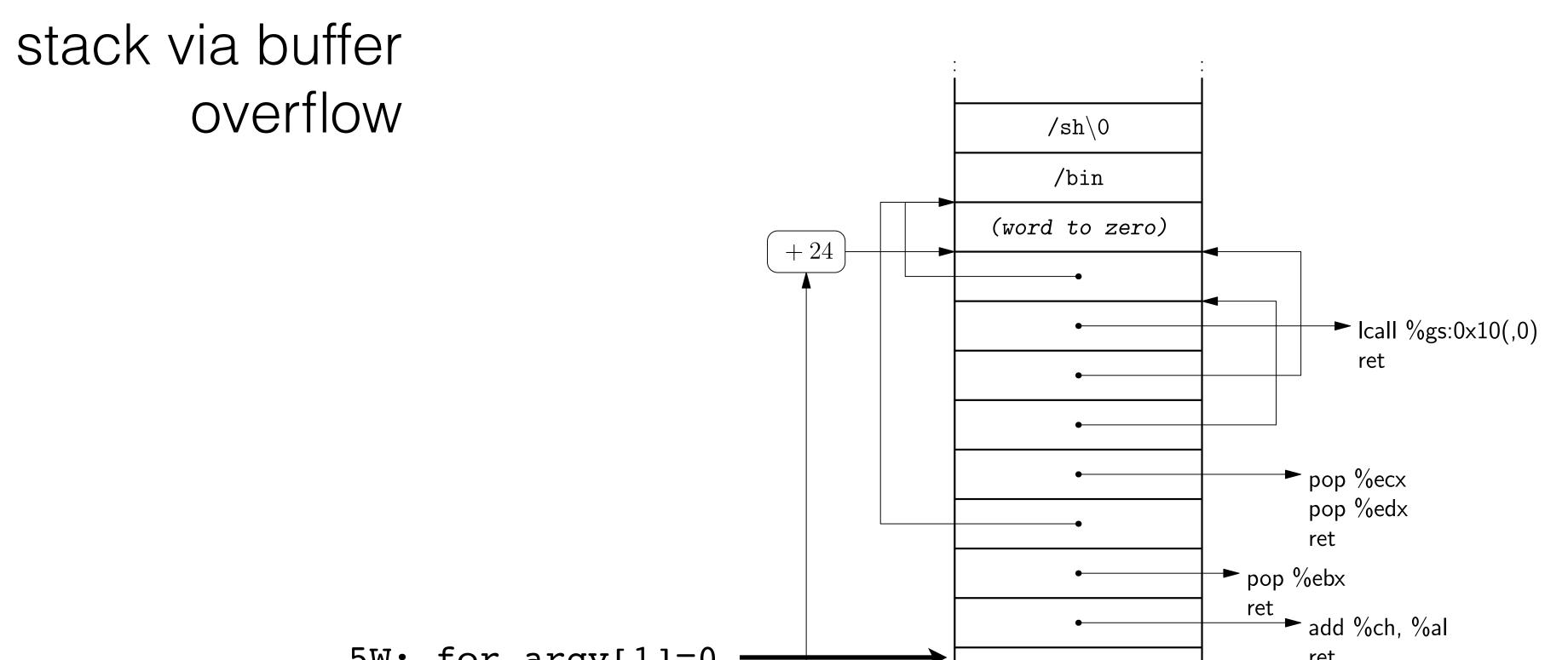
stack via buffer overflow



procedure:

1. call # (0xb) into %eax 2. (ptr. to) program to run ('/bin/sh') in %ebx 3. set argv: pointer to ptr. array ('/bin/sh' and NULL) in %ecx 4. envp: pointer to NULL in %edx

Wirginia Tech
Invent the Future



%esp

5W: for argv[1]=0

2--4W: begin load of

0xb into eax; addr

for argv[1] into edx

1W: eax to zero

The movl %eax, 24(%edx)

Ox0b0b0b0b

Ox0b0b0b0b

The movl %eax, 24(%edx)

The pop %ecx

For argv[1]

The po

1W: eax to zero for argv[I] and call#

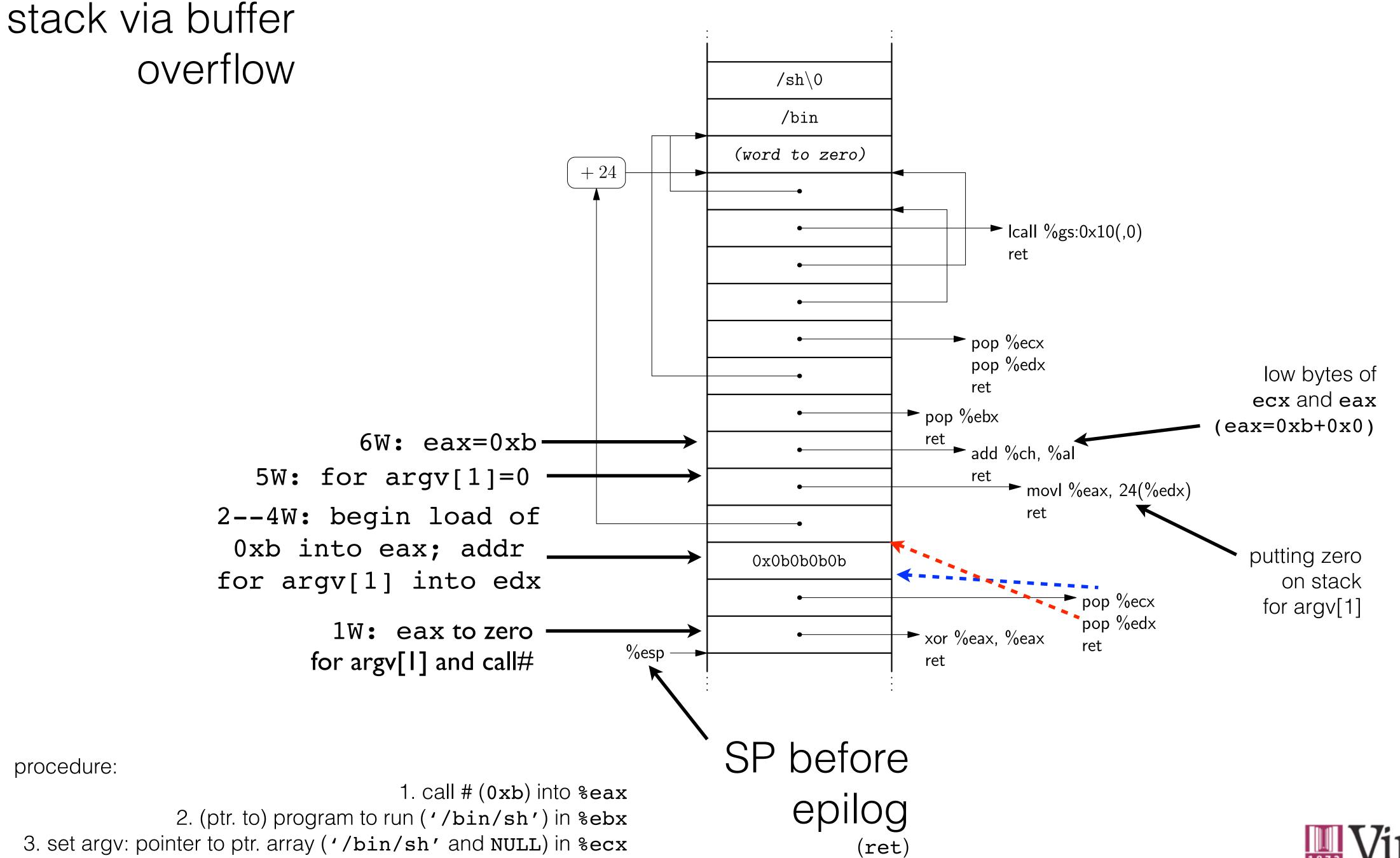
procedure:

1. call # (0xb) into %eax
2. (ptr. to) program to run ('/bin/sh') in %ebx
3. set argv: pointer to ptr. array ('/bin/sh' and NULL) in %ecx
4. envp: pointer to NULL in %edx

SP before epilog

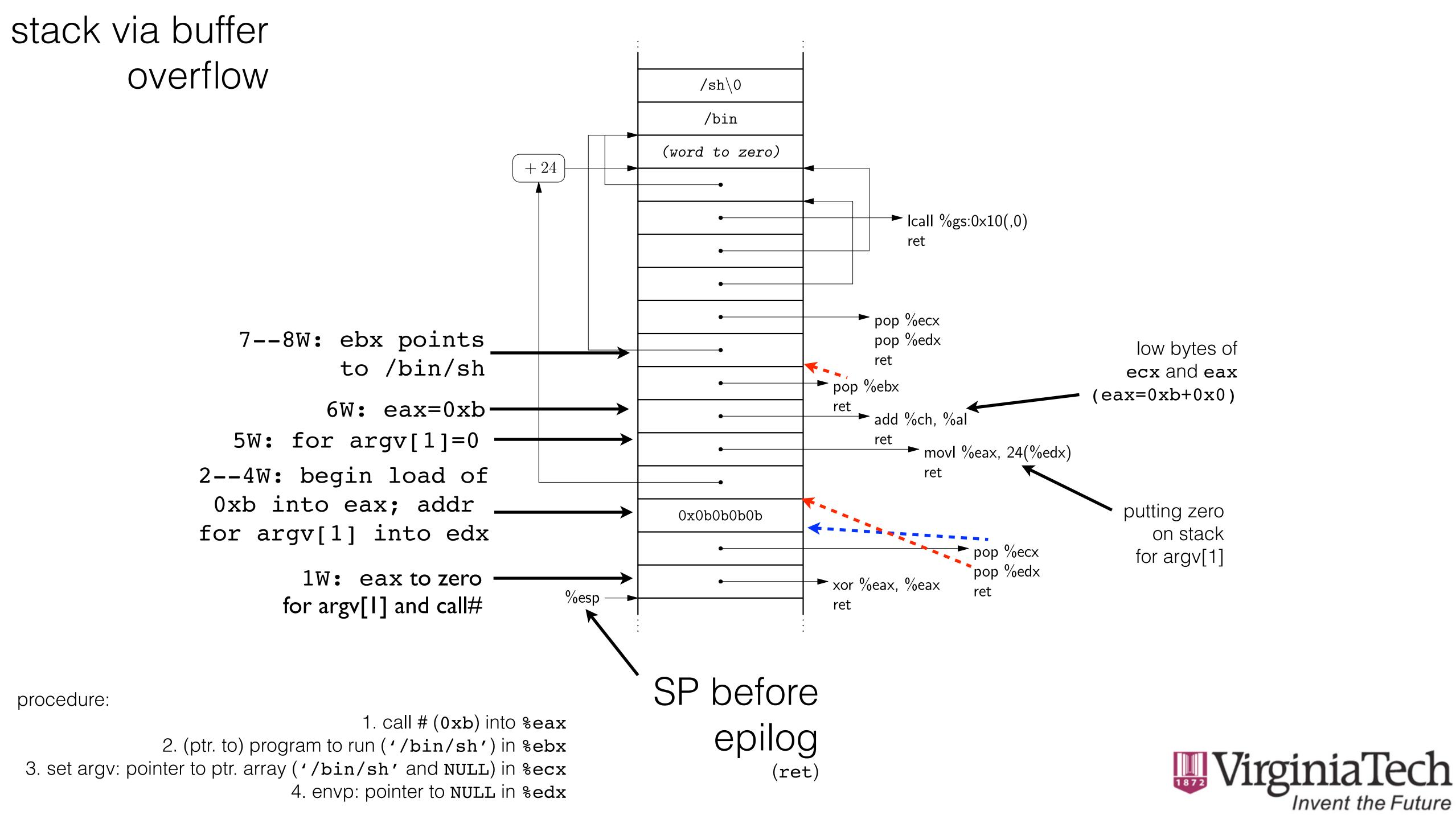
ret

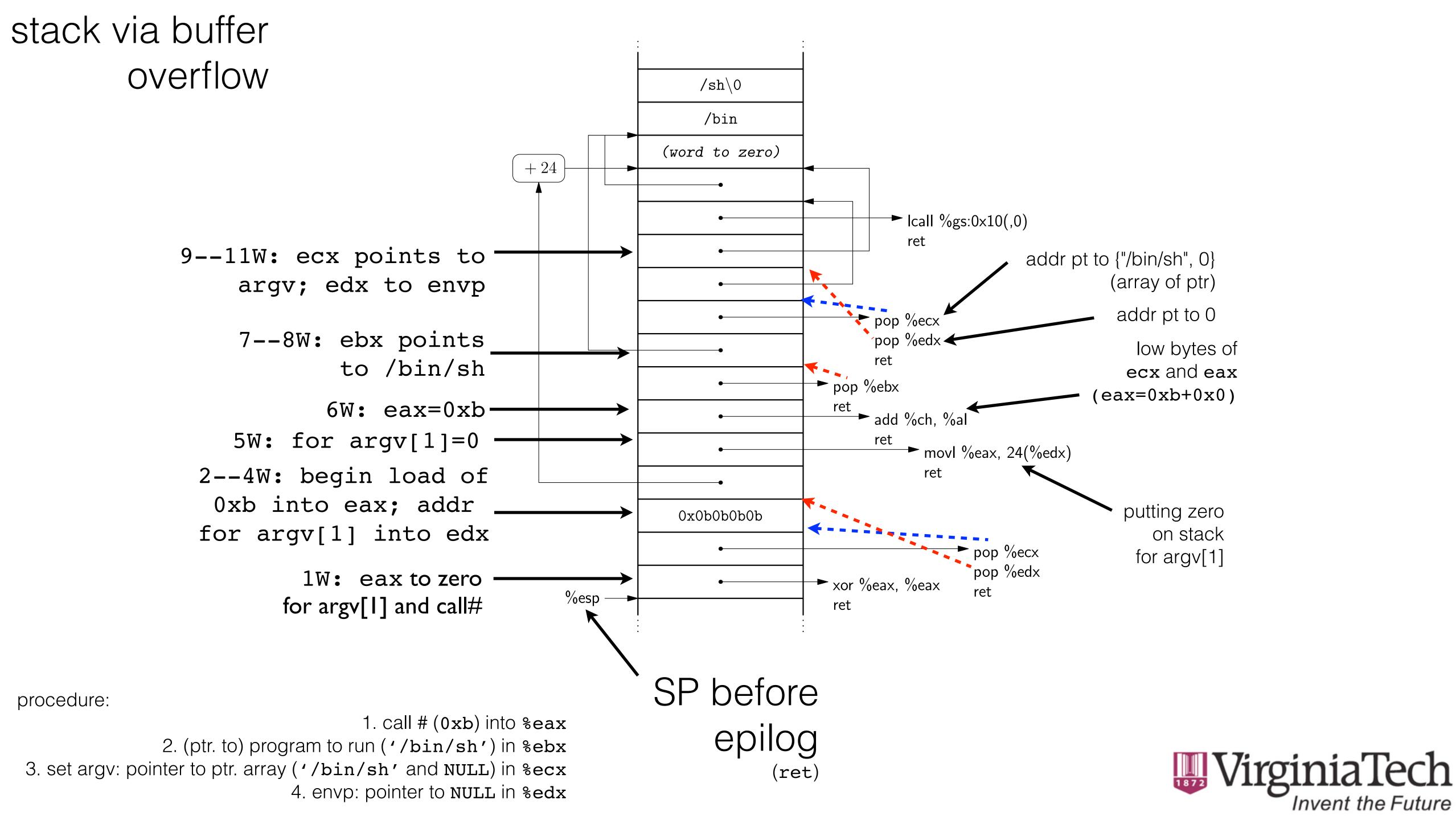




4. envp: pointer to NULL in %edx

Wirginia Tech
Invent the Future





stack via buffer argv: array of ptrs. overflow  $/\text{sh}\setminus 0$ /bin (word to zero) +24► Icall %gs:0x10(,0) 9--11W: ecx points to argv; edx to envp pop %ecx 7--8W: ebx points ′pop %edx∢

to /bin/sh pop %ebx

%esp

6W: eax=0xb

5W: for argv[1]=0

2--4W: begin load of 0xb into eax; addr

for argv[1] into edx

1W: eax to zero for argv[I] and call#

> SP before epilog

0x0b0b0b0b

ret ► add %ch, %al

➤ xor %eax, %eax

ret

(ret)

→ movl %eax, 24(%edx)

procedure:

1. call # (0xb) into %eax 2. (ptr. to) program to run ('/bin/sh') in %ebx 3. set argv: pointer to ptr. array ('/bin/sh' and NULL) in %ecx 4. envp: pointer to NULL in %edx



addr pt to {"/bin/sh", 0}

(array of ptr)

addr pt to 0

eax=0xb+0x0)

putting zero

on stack

for argv[1]

low bytes of

ecx and eax

stack via buffer argv: array of ptrs. overflow  $/\text{sh} \setminus 0$ /bin (word to zero) +2412W: kernel trap. ► Icall %gs:0x10(,0) 9--11W: ecx points to addr pt to {"/bin/sh", 0} argv; edx to envp pop %ecx ' 7--8W: ebx points ′pop %edx∢ to /bin/sh pop %ebx ret ► add %ch, %al 6W: eax=0xb5W: for argv[1]=0

%esp

procedure:

1. call # (0xb) into %eax 2. (ptr. to) program to run ('/bin/sh') in %ebx 3. set argv: pointer to ptr. array ('/bin/sh' and NULL) in %ecx 4. envp: pointer to NULL in %edx

2--4W: begin load of

0xb into eax; addr

for argv[1] into edx

1W: eax to zero

for argv[I] and call#

SP before epilog (ret)

0x0b0b0b0b



(array of ptr)

addr pt to 0

eax=0xb+0x0)

putting zero

on stack

for argv[1]

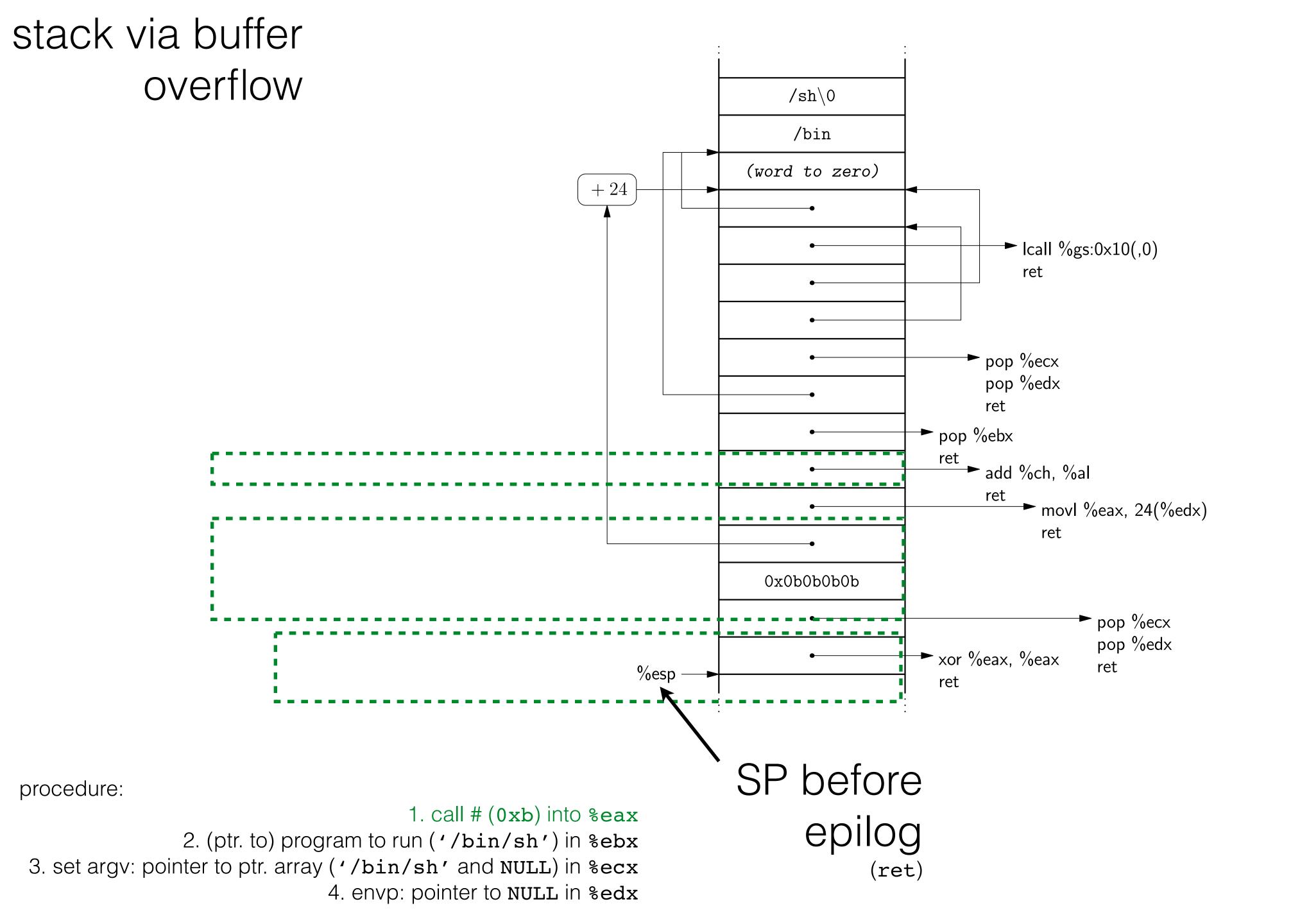
→ movl %eax, 24(%edx)

➤ xor %eax, %eax

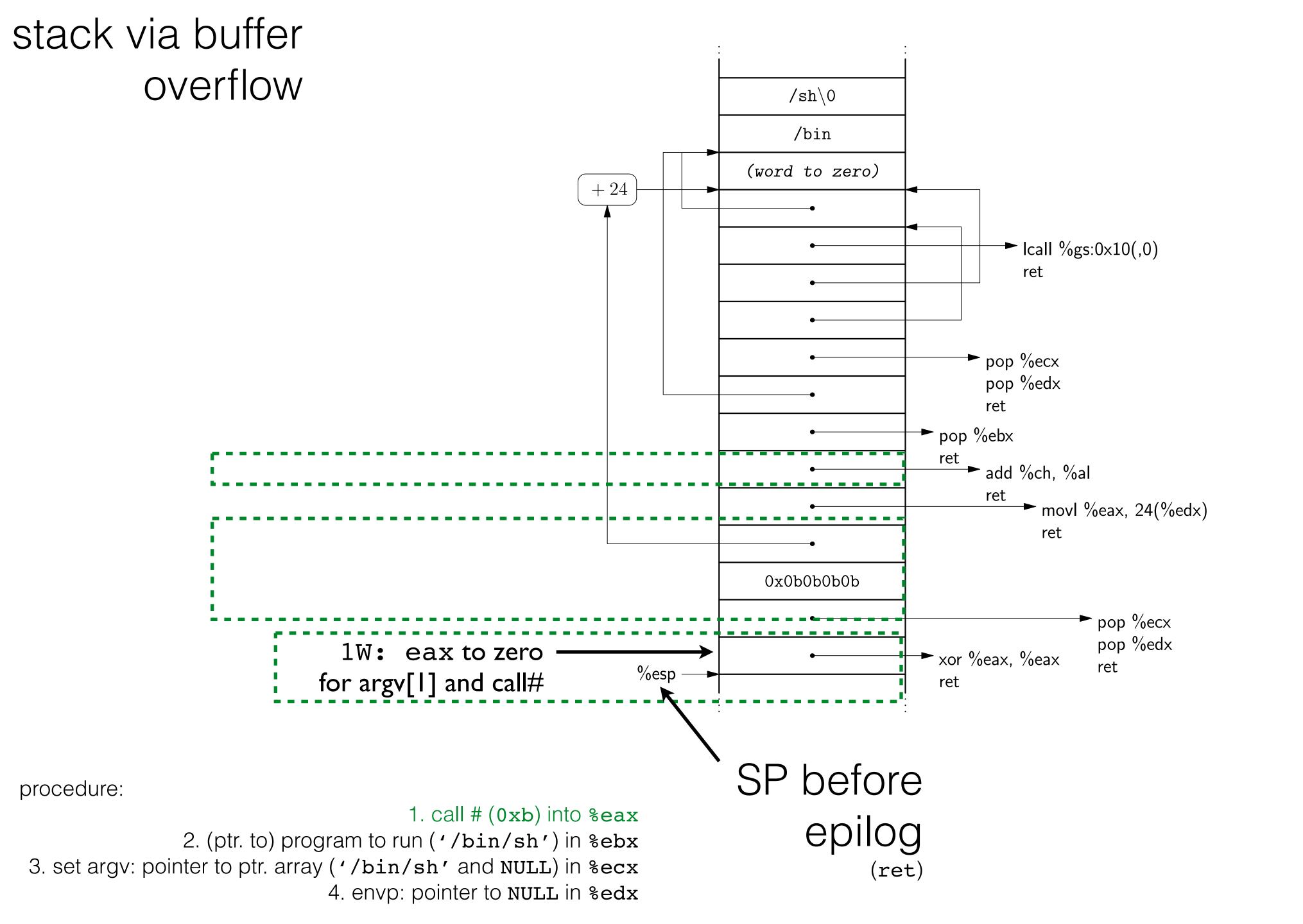
ret

low bytes of

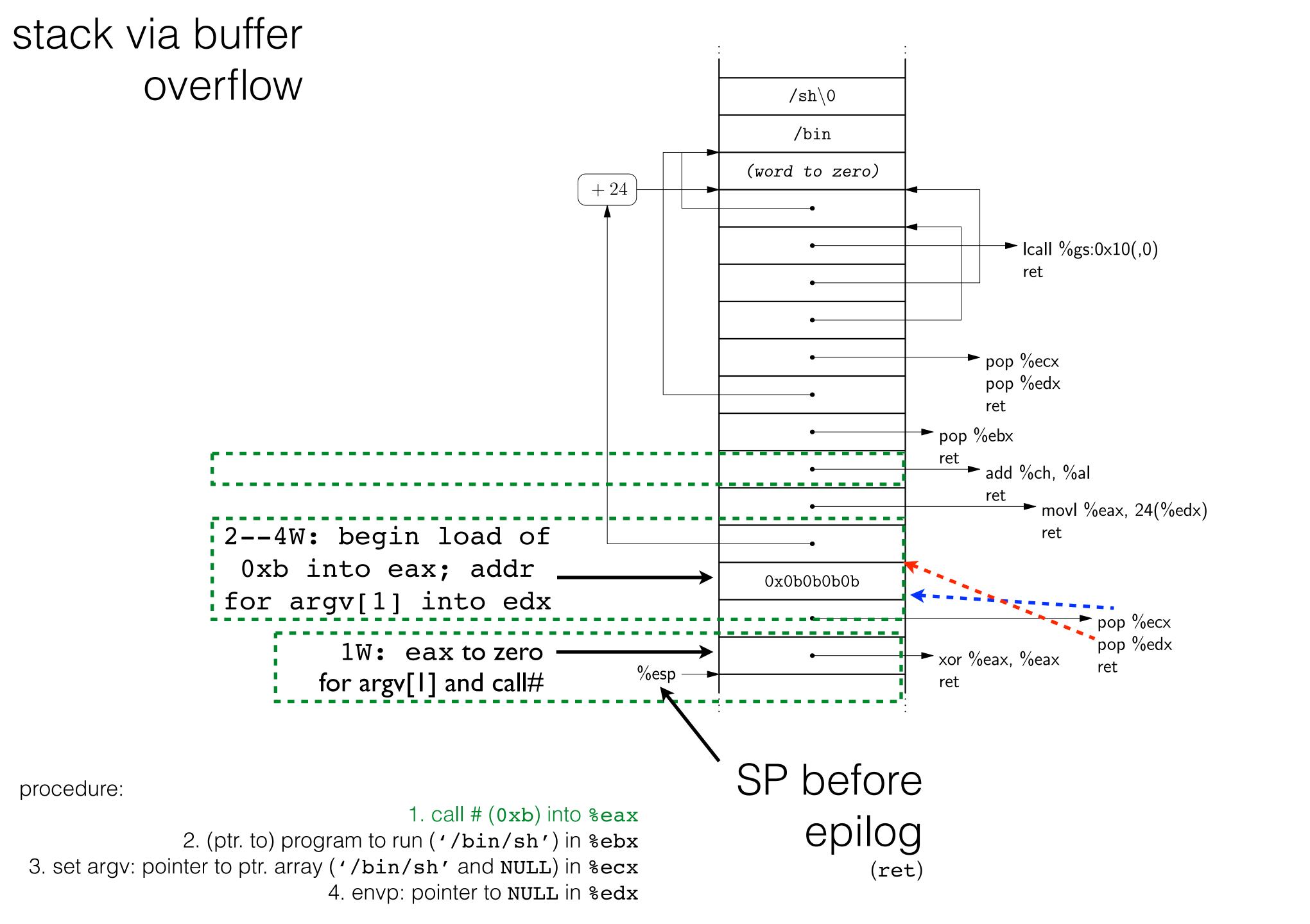
ecx and eax



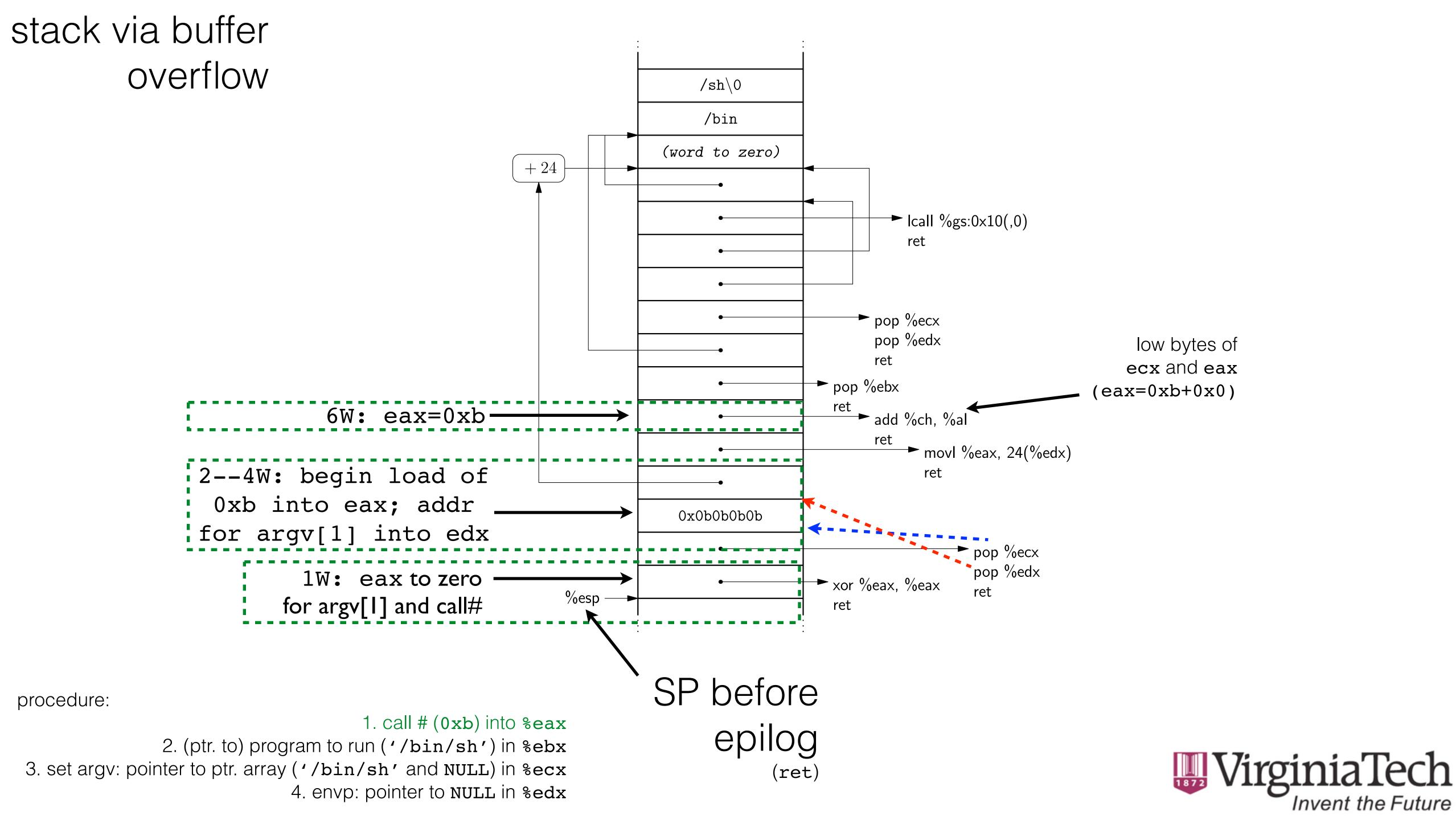


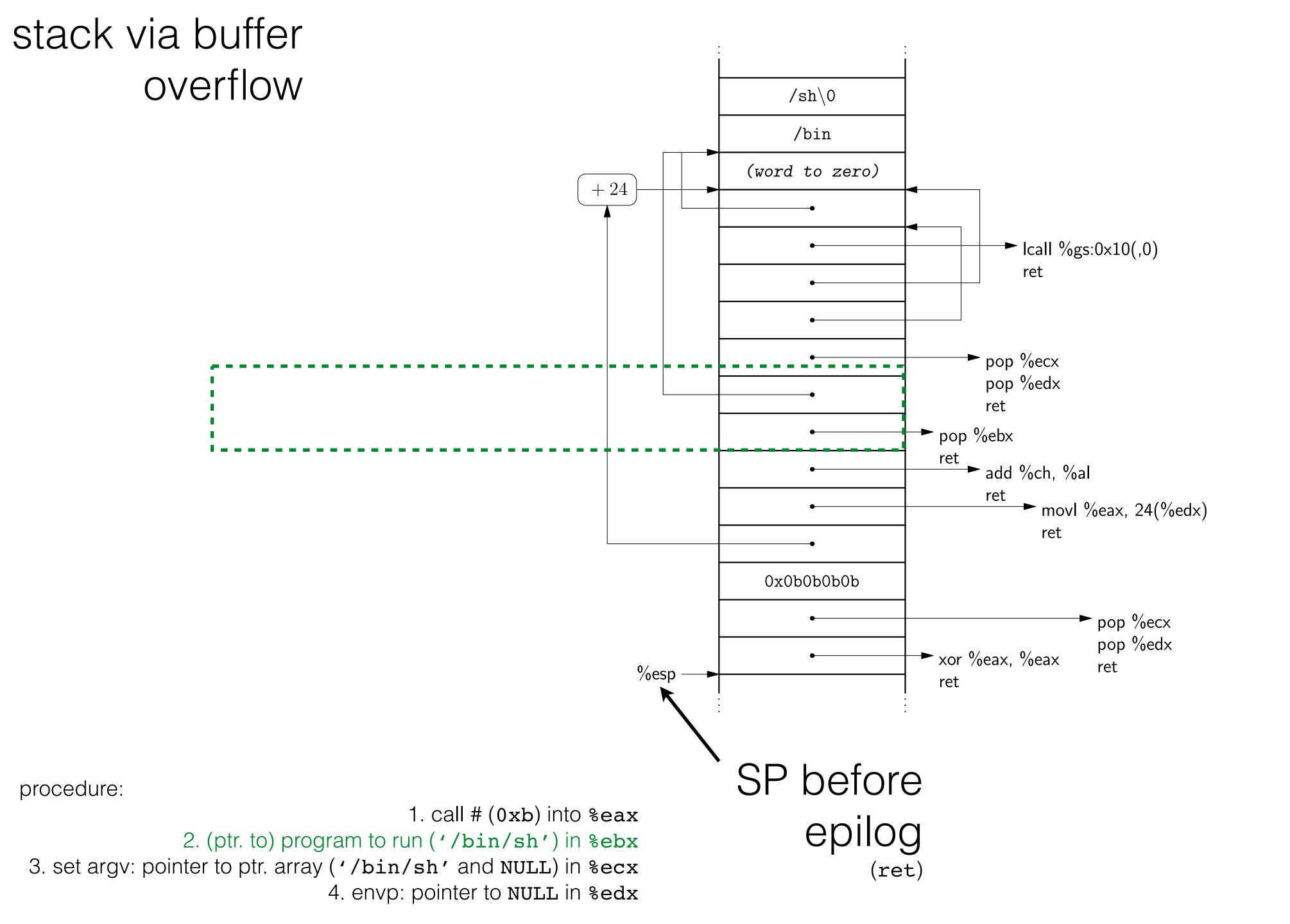




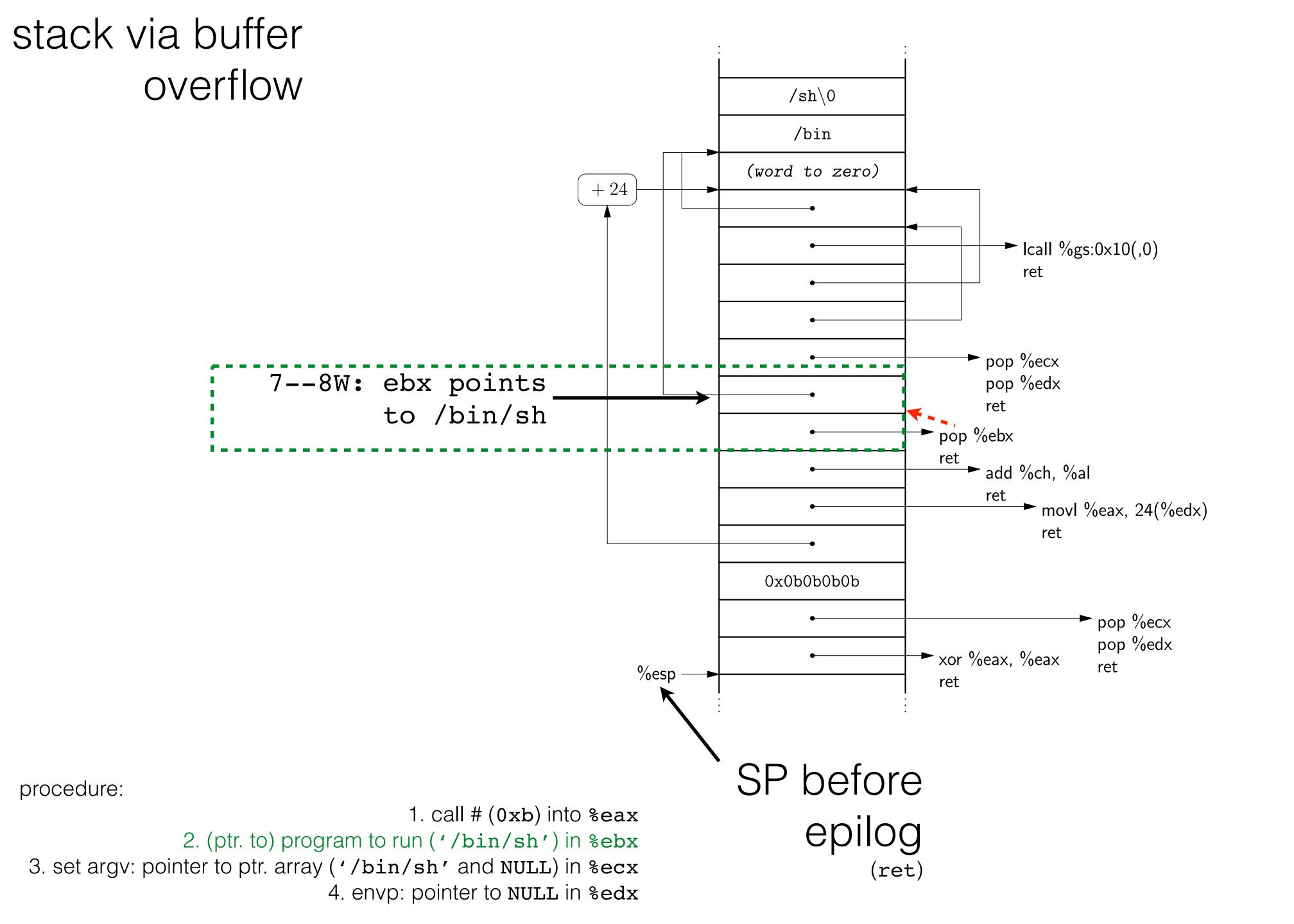




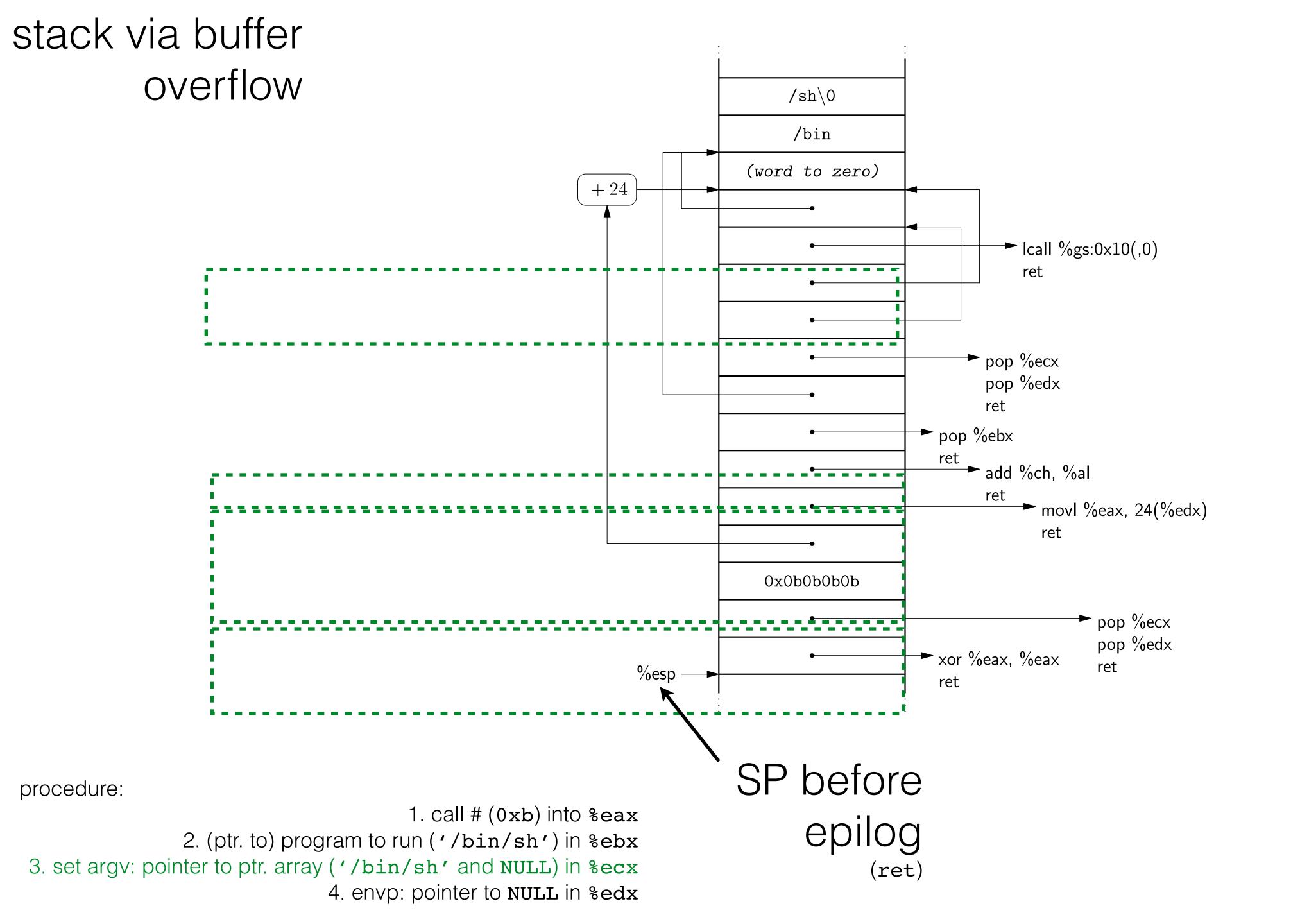




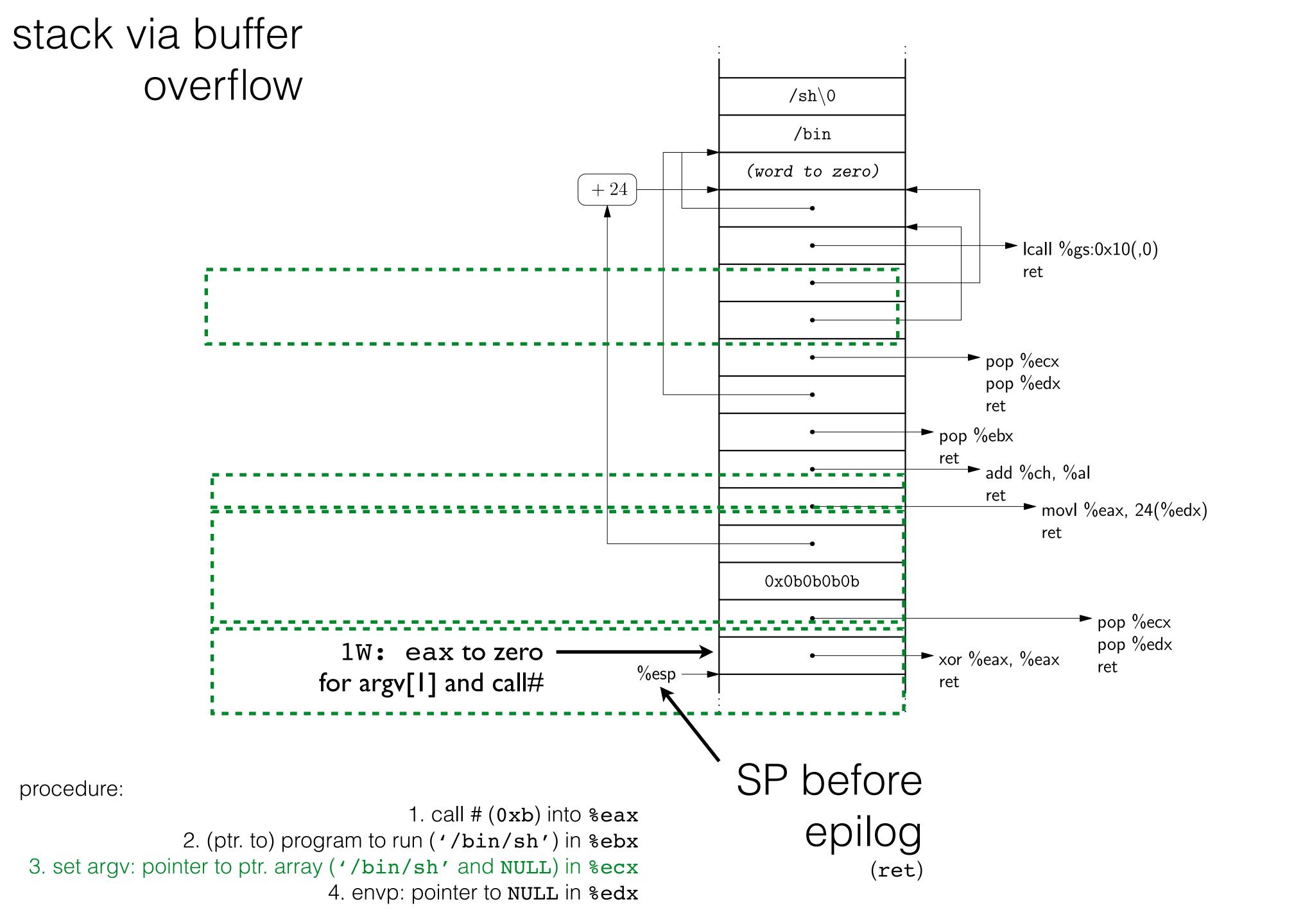




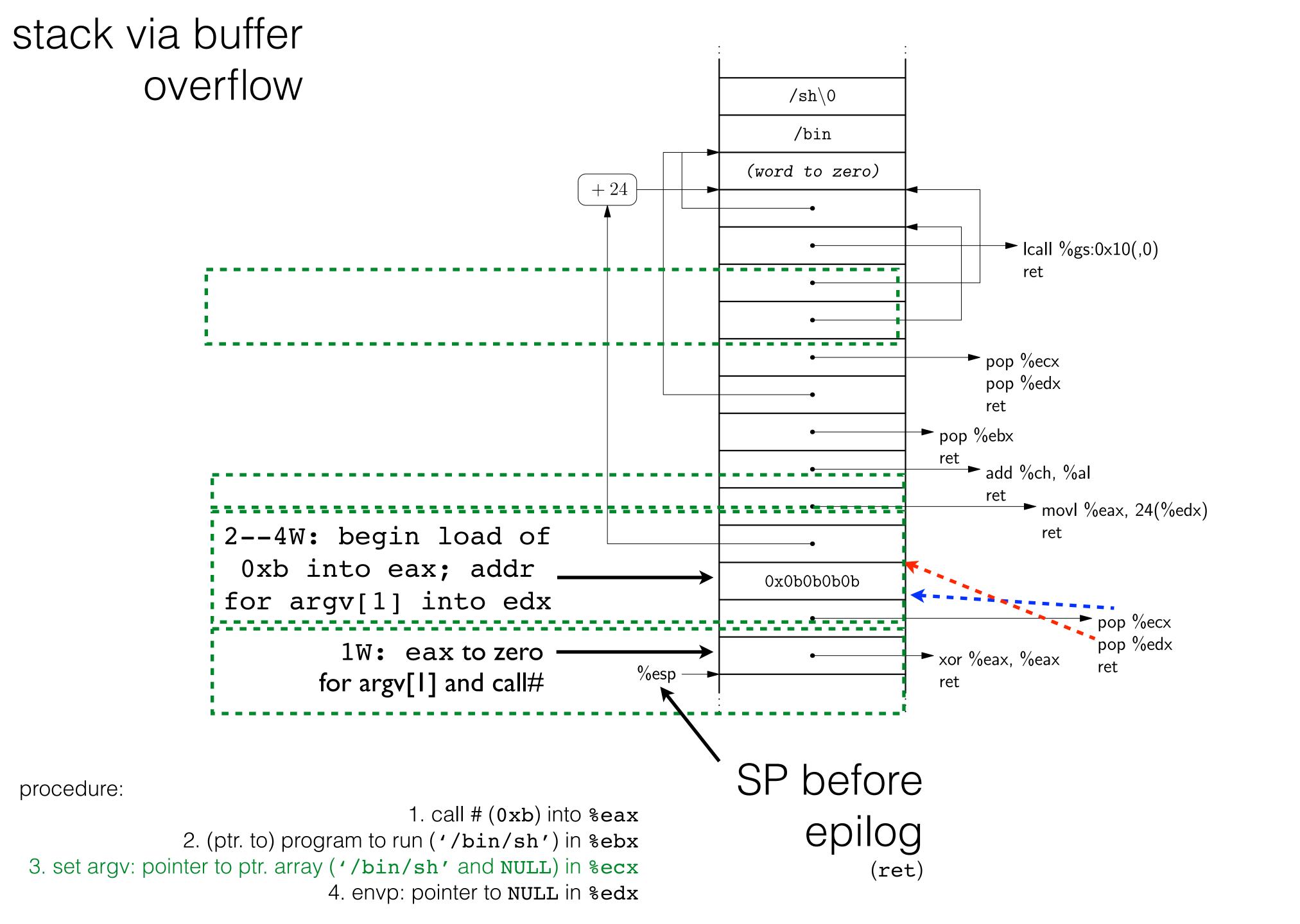




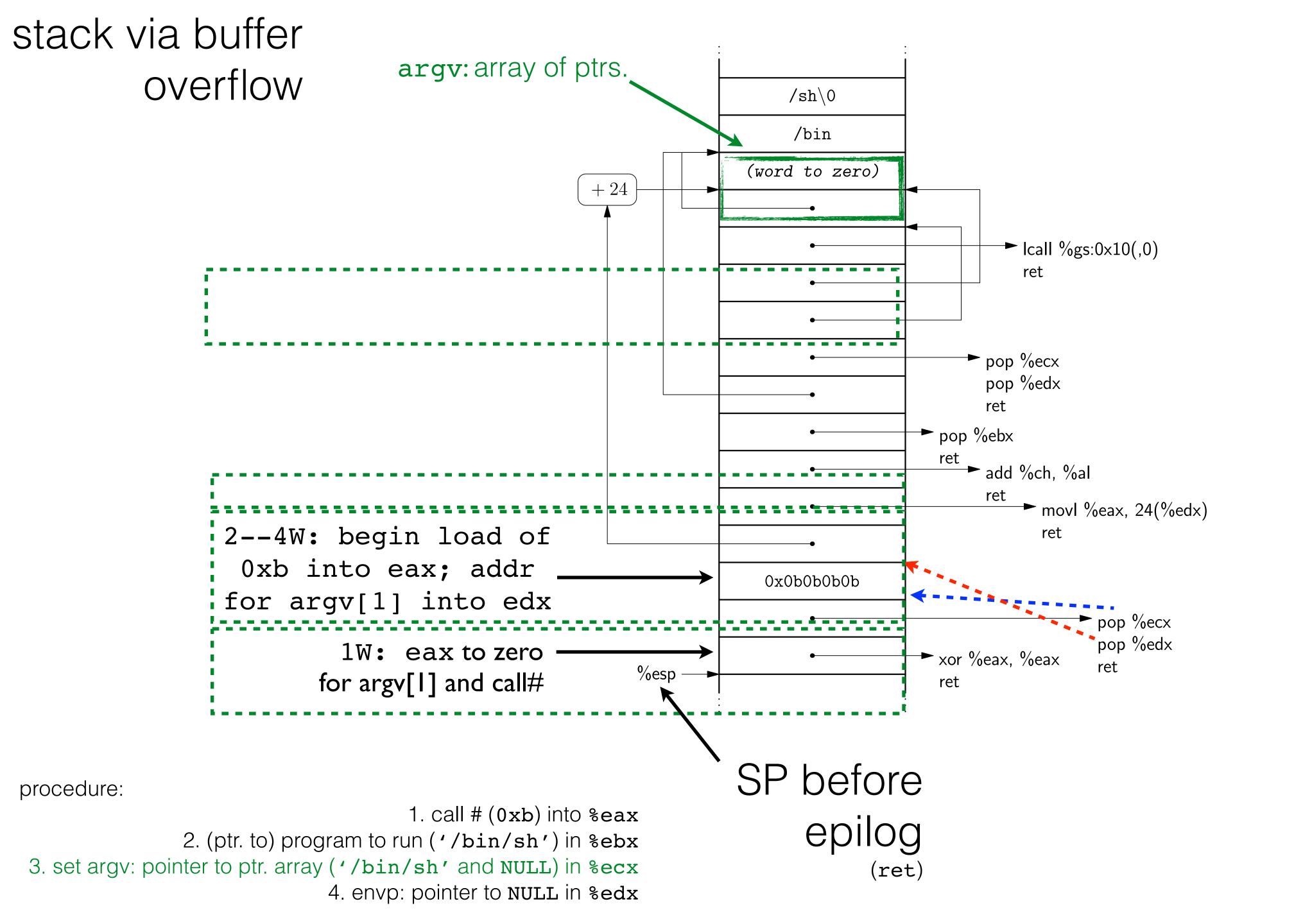




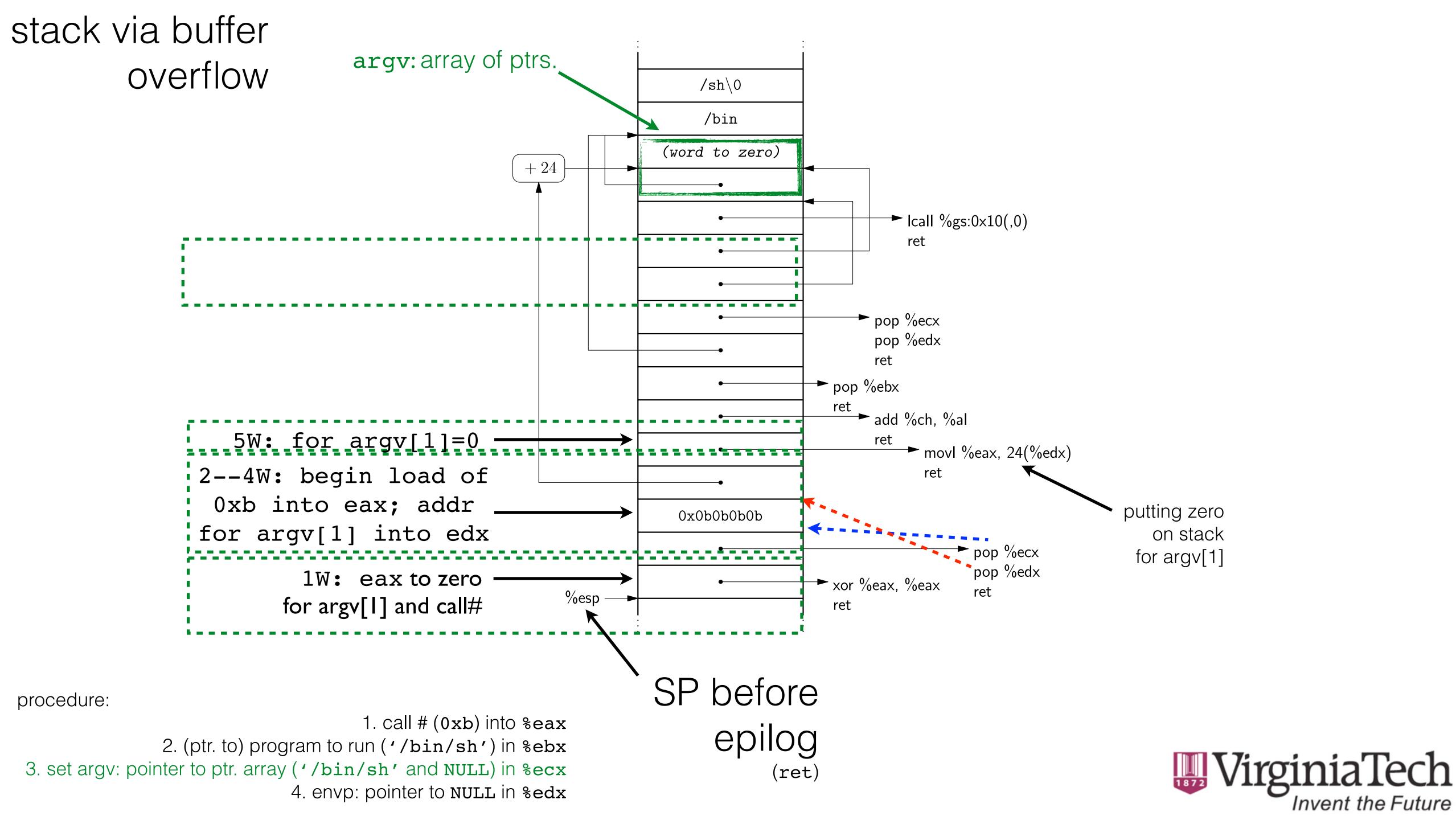


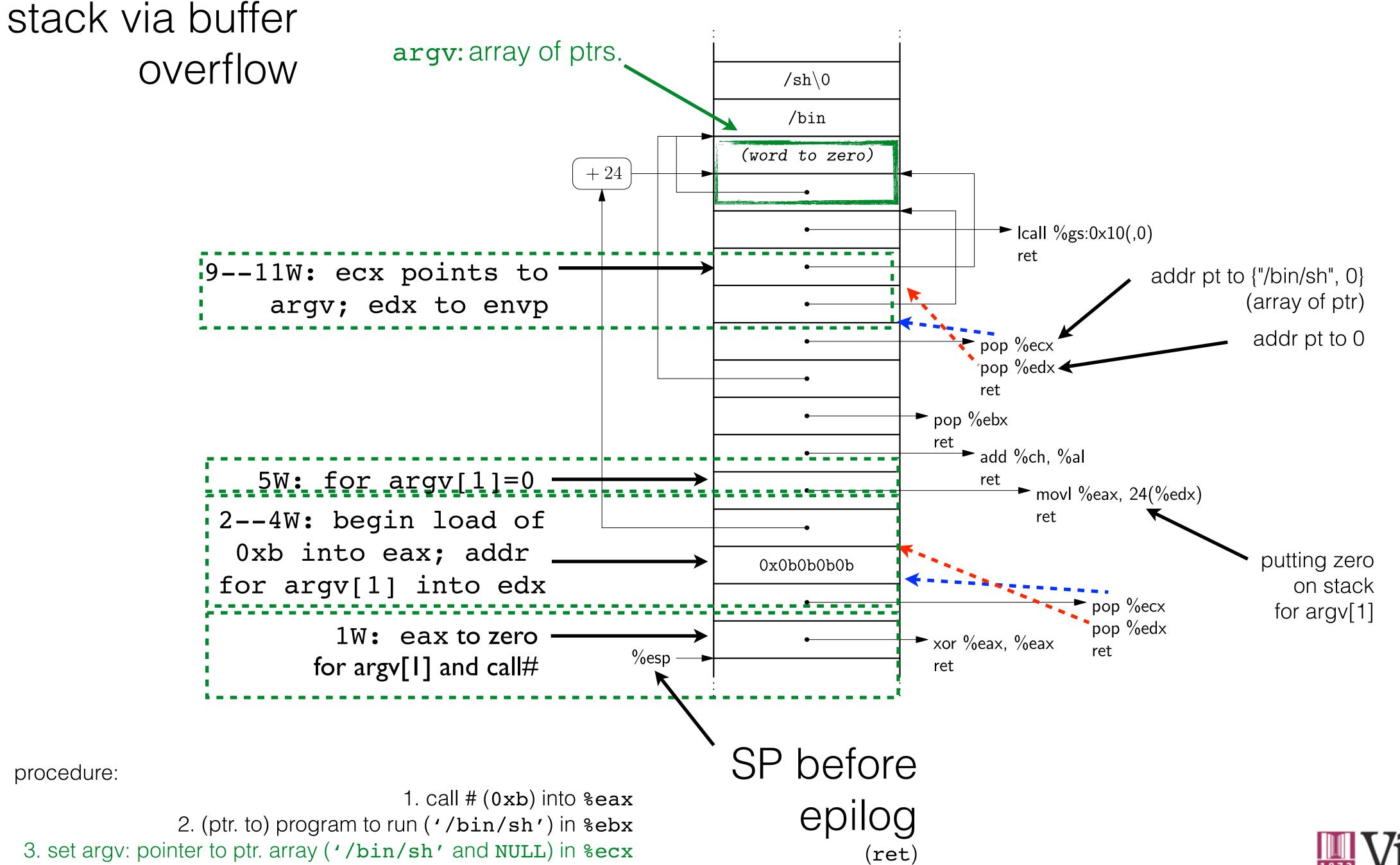






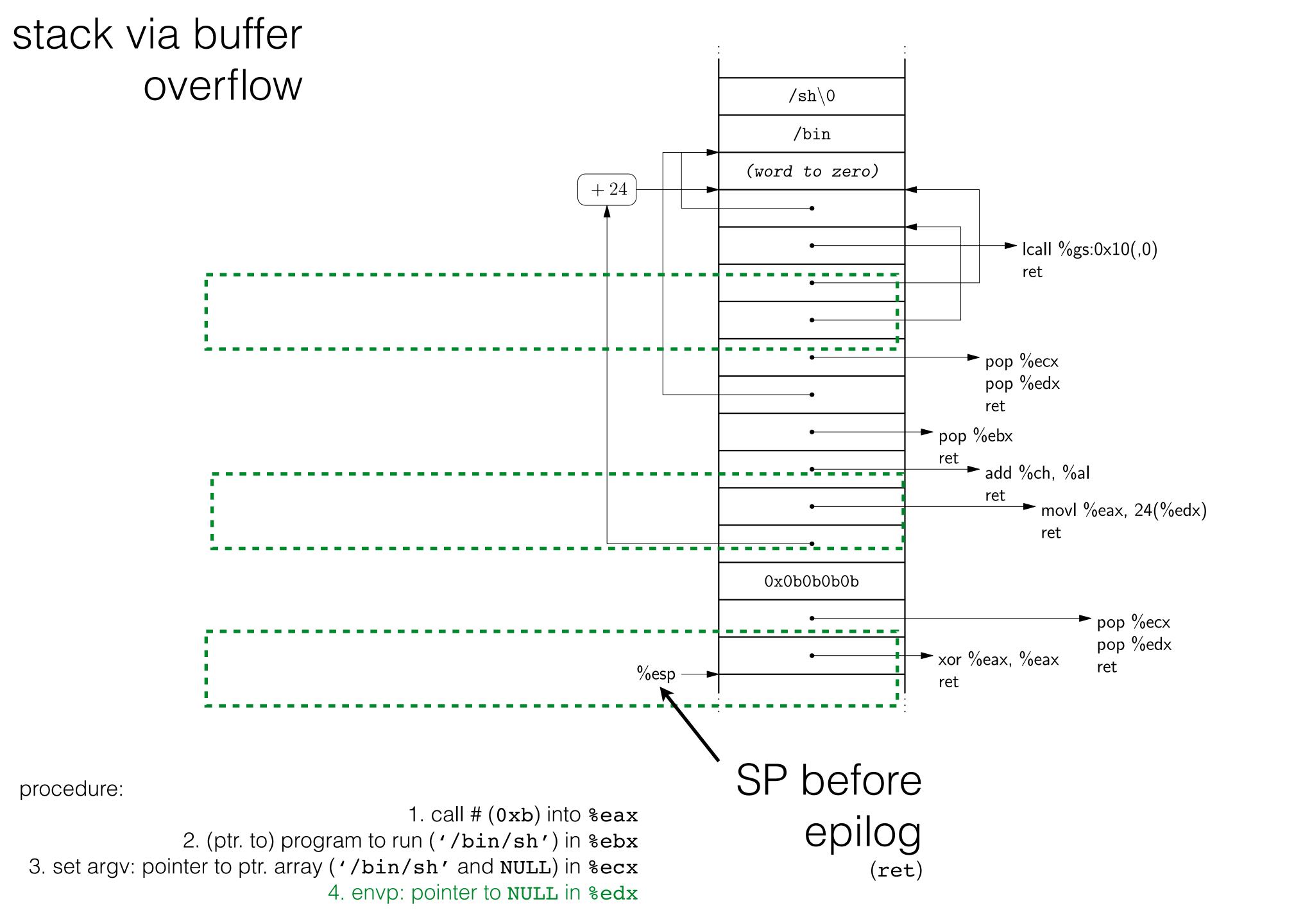




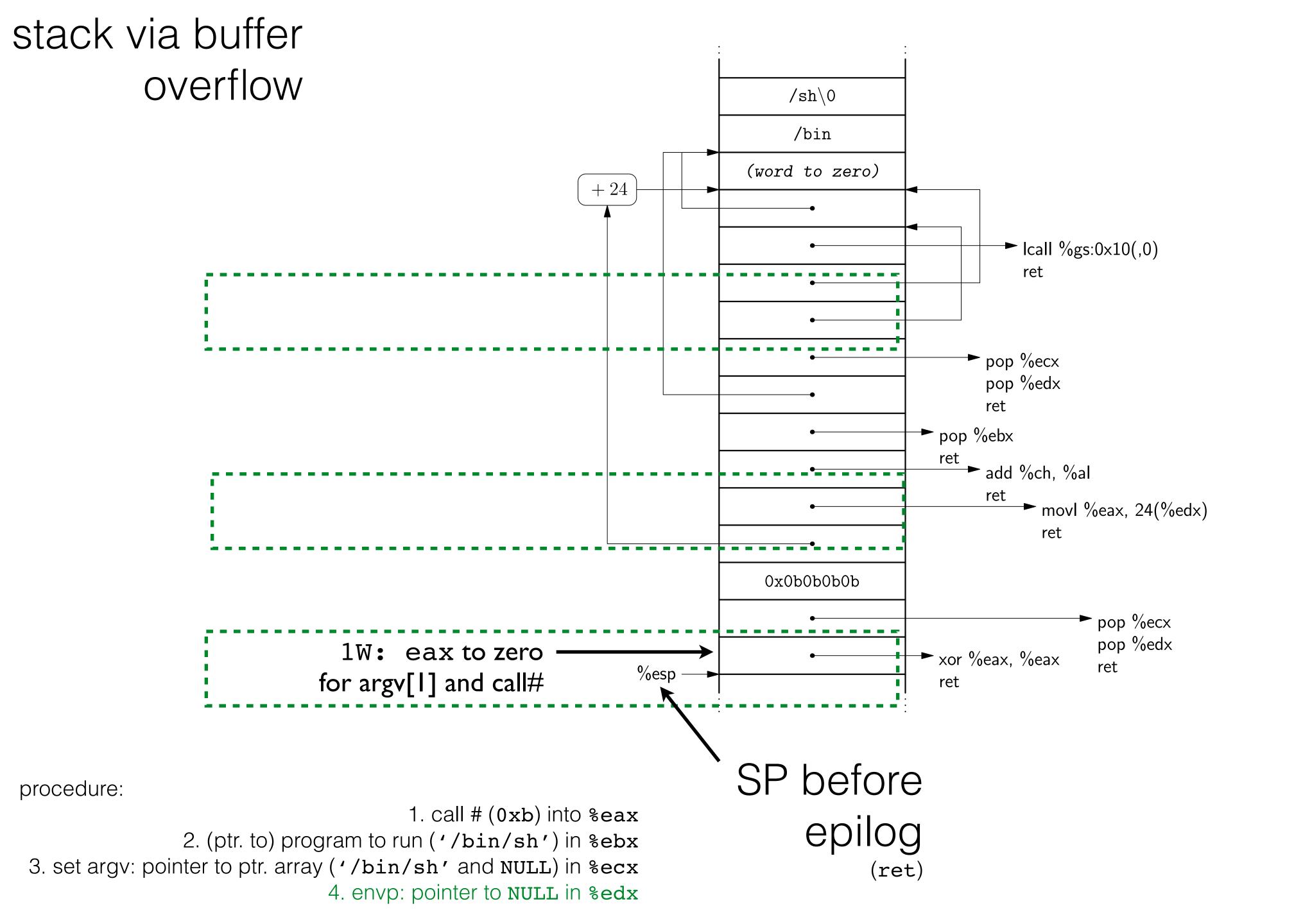


4. envp: pointer to NULL in %edx

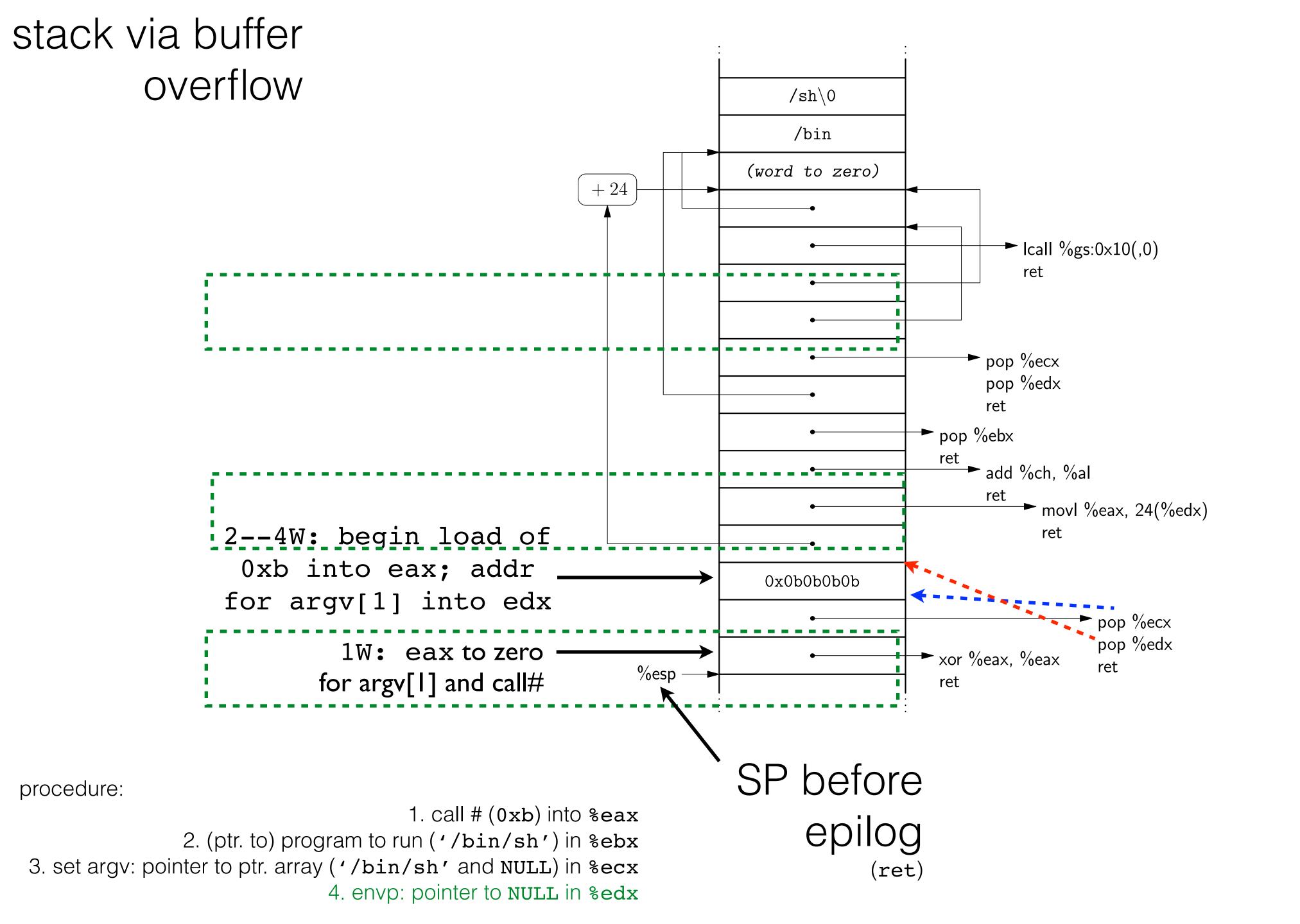
Wirginia Tech Invent the Future



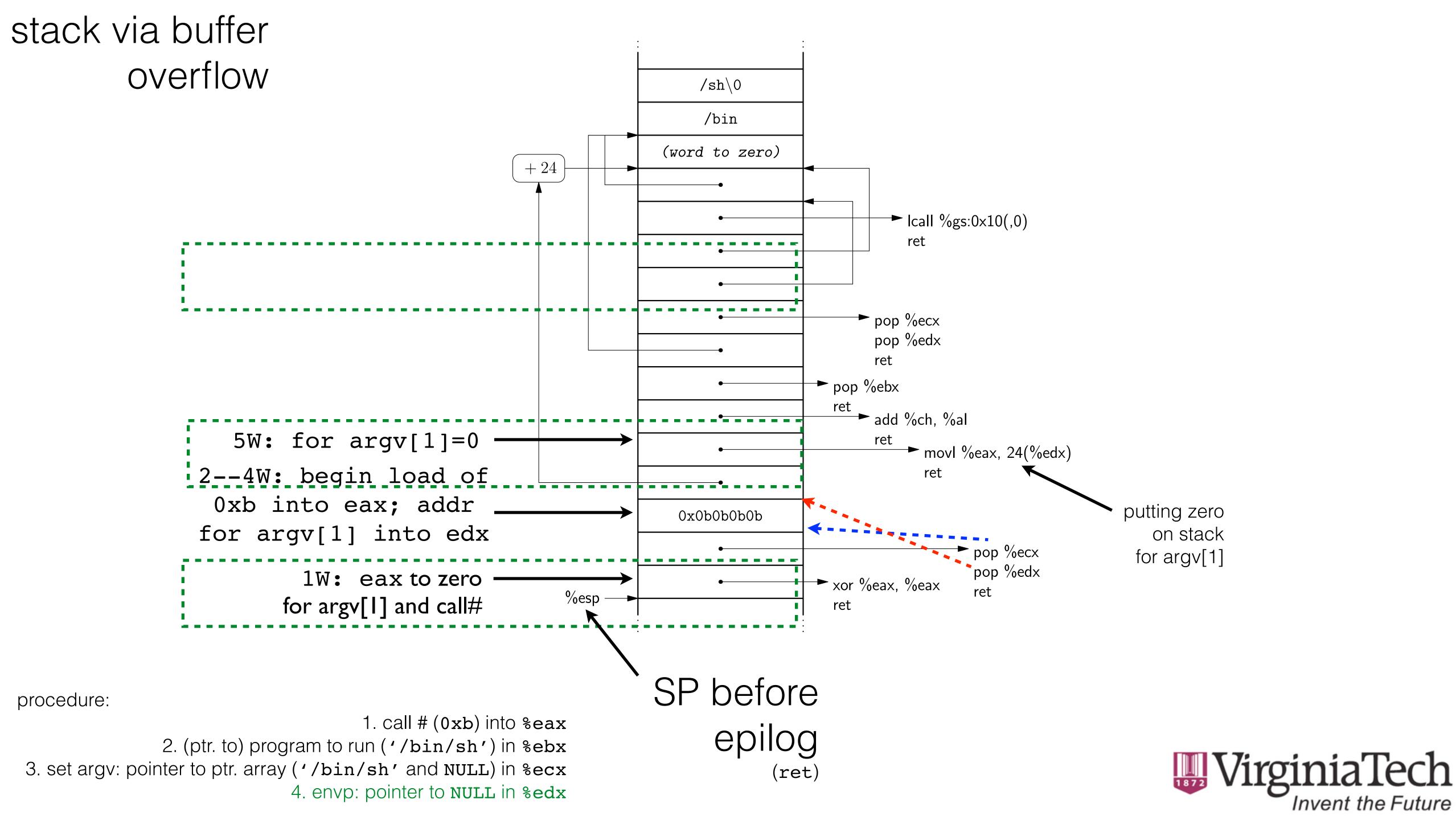


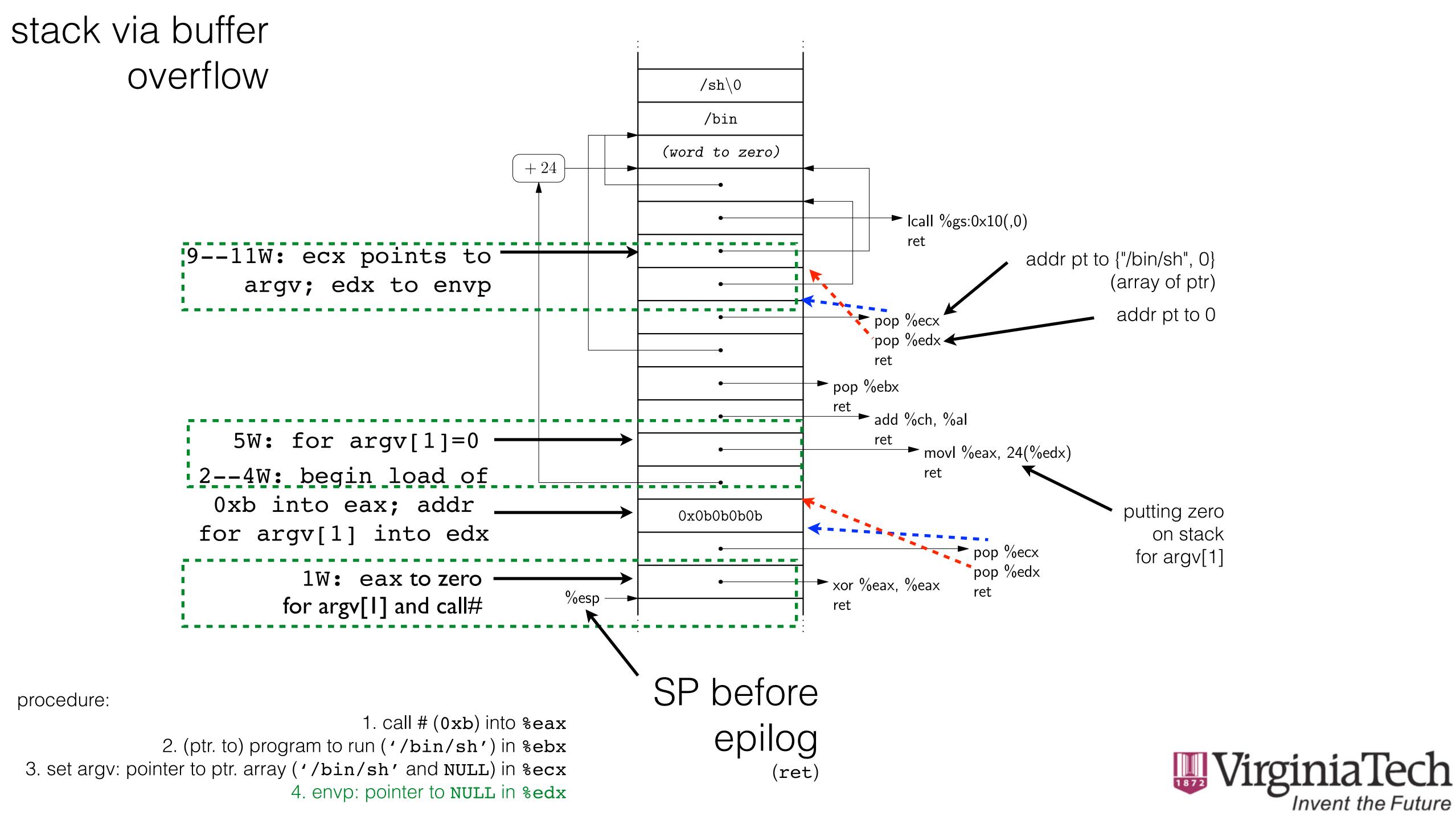












#### stack protection methods:

make the overflow difficult to exploit
 make overflow noticeable
 3.make it impossible



## stack protection methods:

make the overflow difficult to exploit
 make overflow noticeable
 3.make it impossible

1. address space layout randomization 2. NX, W^X



## stack protection methods:

1. make the overflow difficult to exploit

2. make overflow noticeable ←

3.make it impossible

1. address space layout randomization 2. NX, W^X

1. shadow stack2. StackGuard





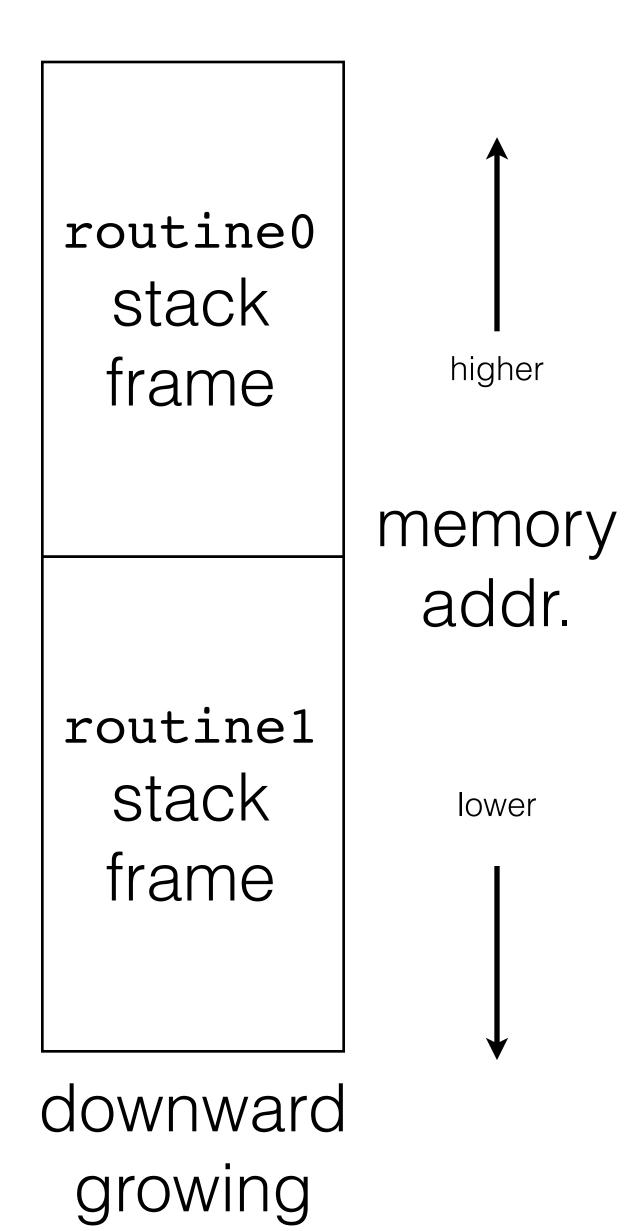
- 1. make the overflow difficult to exploit
  - 2. make overflow noticeable +
    - 3.make it impossible

1. address space layout randomization 2. NX, W^X

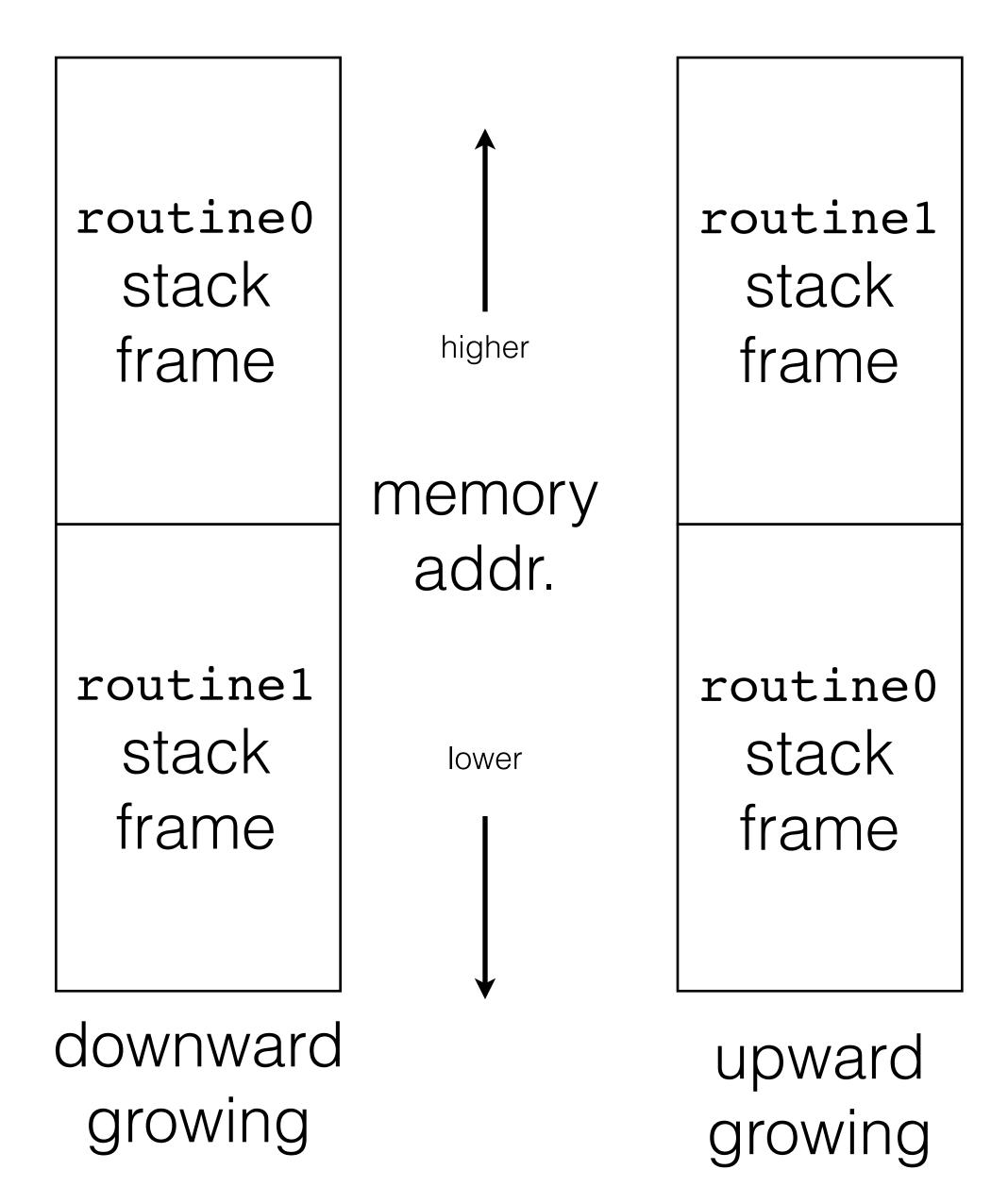
1. shadow stack2. StackGuard

1. separate control and data stack (SCADS)











routine0 routine1 stack stack higher frame frame memory addr. routine1 routine0 stack stack lower frame frame downward

growing

Q: would an upward growing stack prevent exploit of buffer overflow?

upward growing



## upward growing stack buffer overflow

```
void function(char *str) {
    char buffer[16];

    strcpy(buffer,str);
}

void main() {
    char large_string[256];
    int i;

for( i = 0; i < 255; i++)
        large_string[i] = 'A';

function(large_string);
}</pre>
```



## upward growing stack buffer overflow

```
void function(char *str) {
  char buffer[16];
  strcpy(buffer,str);
void main() {
 char large_string[256];
  int i;
 for( i = 0; i < 255; i++)
   large string[i] = 'A';
 function(large_string); function
                                                   points to str
```



### upward growing stack buffer overflow

```
void function(char *str) {
  char buffer[16];
                                                        points to buffer
  strcpy(buffer,str);
void main() {
  char large string[256];
  int i;
                                            buffer ]
 for( i = 0; i < 255; i++)
   large string[i] = 'A';
 function(large_string);
                                                    points to str
```

### upward growing stack buffer overflow

```
void function(char *str) {
  char buffer[16];
                                                         points to buffer
  strcpy(buffer,str);
void main() {
  char large string[256];
  int i;
  for( i = 0; i < 255; i++)
   large string[i] = 'A';
 function(large_string);
                                                     points to str
```

### upward growing stack buffer overflow

```
[???]

[sfp]

[argl]

[arg2]

points to buffer

[arg2]

points to str

[buffer]

[sfp]

[sfp]

[ret]

[argl]
                                                                        to exploit: make
                                                                        overwrite ret of
                                                                                    strcpy
                                         points to str
```

void function(char \*str) {

strcpy(buffer,str);

char buffer[16];



motivation: attacker must know address of libraries/instructions to return to



motivation: attacker must know address of libraries/instructions to return to

PIE	Main Executable	Heap	Stack	Shared Libraries	Linker
No	Fixed	Randomized per execution	Fixed	Randomized per device boot	Fixed
Yes	Randomized per execution	Randomized per execution	Randomized per execution	Randomized per device boot	Randomized per execution



motivation: attacker must know address of libraries/instructions to return to

address of X, ————
randomised

PIE	Main Executable	Heap	Stack	Shared Libraries	Linker
No	Fixed	Randomized per execution	Fixed	Randomized per device boot	Fixed
Yes	Randomized per execution	Randomized per execution	Randomized per execution	Randomized per device boot	Randomized per execution



motivation: attacker must know address of libraries/instructions to return to

position independent executable/code

PIE	Main Executable	Heap	Stack	Shared Libraries	Linker
No	Fixed	Randomized per execution	Fixed	Randomized per device boot	Fixed
Yes	Randomized per execution	Randomized per execution	Randomized per execution	Randomized per device boot	Randomized per execution



#### shadow stack

routine0 stack frame

routine1 stack frame

working stack (overflow occurs here)



#### shadow stack

routine0 stack frame

routine1 stack frame routine0 stack frame

routine1 stack frame

shadow stack (no process access)

working stack (overflow occurs here)



#### shadow stack

routine0 stack frame

routine0 stack frame

upon ret check if return addresses match

routine1 stack frame

routine1 stack frame

shadow stack (no process access)

working stack (overflow occurs here)



# StackGuard (canary)

```
[ret]
function
[sfp]
[buffer]
```



### StackGuard (canary)



### StackGuard (canary)





parameters
RIP #1
SFP #1
saved data registers
local variables and buffers
parameters
RIP #2
SFP #2
 saved data registers
 local variables and buffers

(a) Single call stack.



parameters
RIP #1
SFP #1
saved data registers
local variables and buffers
parameters
RIP #2
SFP #2
saved data registers

(a) Single call stack.

RIP #1
SFP #1
RIP #2
SFP #2

	parameters		
	saved data registers		
loc	cal variables and buffers		
parameters			
	saved data registers		
loc	cal variables and buffers		

(b) Separated CS and DS.



naramatara	
parameters	
RIP #1	
SFP #1	
saved data registers	
local variables and buffers	
parameters	
RIP #2	
SFP #2	
saved data registers	
local variables and buffers	

(a) Single call stack.

RIP #1	
SFP #1	
RIP #2	
SFP #2	

(b) Separated CS and DS.

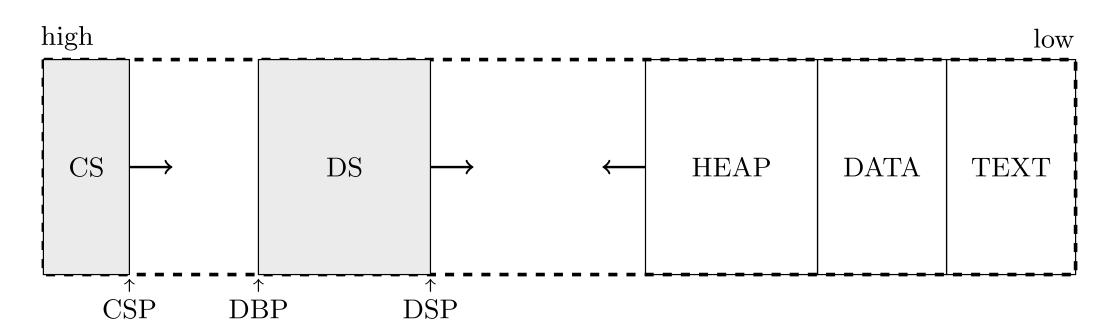


Fig. 2. Virtual address space layout of a user mode process compiled with SCADS.

