

CS356 Unit 7

Data Layout &
Intermediate Stack Frames

Structs

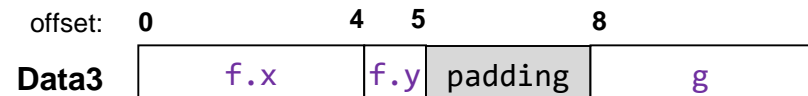
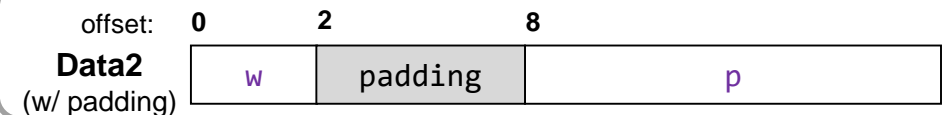
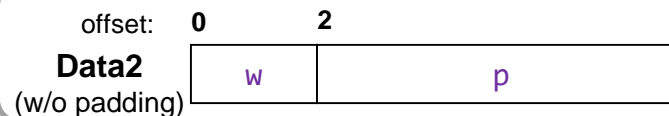
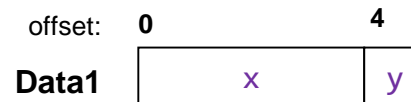
CS:APP 3.9.1

- Structs are just **collections of _____ data**
- Each member is laid out in consecutive memory locations, with some _____ **inserted to ensure alignment**
 - Intel machines don't require alignment but perform better when it is used
 - Reordering can reduce size! www.catb.org/esr/structure-packing
 - **“Each type aligned at a multiple of its size”**

```
struct Data1 {
    int x;
    char y;
};

struct Data2 {
    short w;
    char *p;
};

struct Data3 {
    struct Data1 f;
    int g;
};
```



Structs: Offsets in assembly

```
struct record_t {
    char a[2];
    int b;
    long c;
    int d[3];
    short e;
};

void initialize(struct record_t *x) {
    x->a[1] = 1;
    x->b     = 2;
    x->c     = 3;
    x->d[1] = 4;
    x->e     = 5;
}
```

a	a			b	b	b	b
c	c	c	c	c	c	c	c
d0	d0	d0	d0	d1	d1	d1	d1
d2	d2	d2	d2	e	e		

Assume 4-byte **int** / **float**,
8-byte **long** / **double**.

Can you figure out the
offsets for **%rdi** ?

```
initialize:
    movb    $1, __(%rdi)
    movl    $2, __(%rdi)
    movq    $3, __(%rdi)
    movl    $4, __(%rdi)
    movw    $5, __(%rdi)
    ret
```

```
struct B { // this struct must start/end at a multiple of 4, because that's required by 'y'
    char x; // 1 byte
    int y; // 4 bytes (needs 3 bytes of padding before to start at a multiple of 4)
    char z; // 1 byte (needs 3 bytes of padding after to end at a multiple of 4)
};
```

```
struct A {
    char a; // 1 byte
    struct B b; // has 4-byte alignment: 3 bytes of padding before 'b'
    char c; // also 3 bytes of padding before 'c', so that 'b' ends at a multiple of 4
};
```

```
void init(struct A *a) {
    a->a = 1;
    a->b.x = 2;
    a->b.y = 3;
    a->b.z = 4;
    a->c = 5;
}
```

a				x			
y	y	y	y	z			
c							

```
$ gcc -fomit-frame-pointer -mno-red-zone -Og -S align.c; cat align.s | grep mov
```

```
movb    $1, (%rdi)
movb    $2, 4(%rdi)
movl    $3, 8(%rdi)
movb    $4, 12(%rdi)
movb    $5, 16(%rdi)
```

We still want each member of the nested struct to start at a multiple of its size, but where should the nested struct itself start?

Its start/end should have the largest alignment required by its members.

Unions

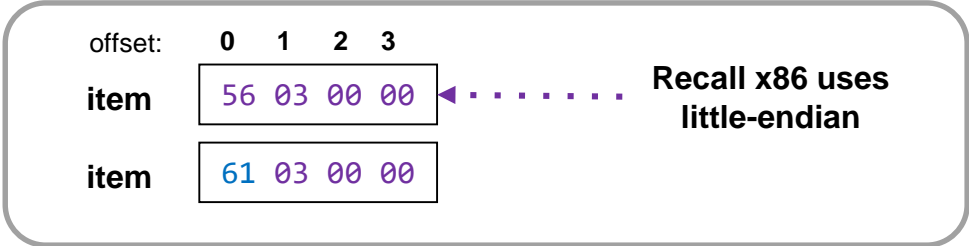
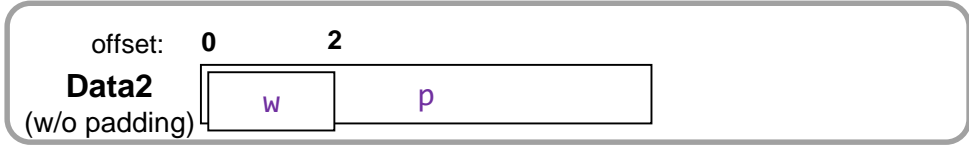
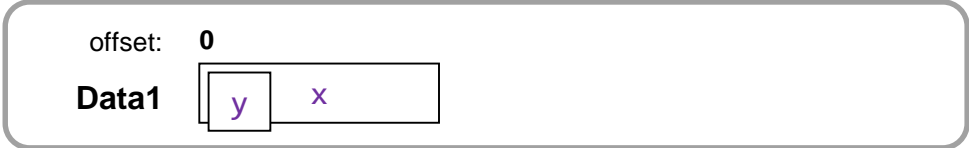
- Unions allow you to read/write the **same** _____ **region** as **variables with** _____ **types**
 - All elements start at offset `__`
 - The size of the union is simply the size of the _____ member
 - Elements must be POD (plain old data) or at least default-constructible

```

union Data1 {
    int x;
    char y;
};

union Data2 {
    short w;
    char *p;
};

int main() {
    union Data1 item;
    item.x = 0x356;
    item.y = 'a';
}
    
```



Unions: Revealing Endianness

```
#include <stdio.h>

union int_bytes {
    int x;
    char bytes[4];
};

int main() {
    union int_bytes ib;
    ib.x = 256;
    printf("%08X is %02X %02X %02X %02X\n",
        ib.x, ib.bytes[3], ib.bytes[2],
        ib.bytes[1], ib.bytes[0]);
}

// prints:
// 00000100 is 00 00 01 00
```

- 4-byte union
- **x** reads/writes an **int**
- **bytes** reads/writes 4 consecutive **char**

Note that bytes are stored in reversed order

Unions: hex encoding of a float

```
#include <stdio.h>

union float_int {
    float f;
    int i;
};

int main() {
    union float_int fi;
    fi.f = 1.0;
    printf("%.2f is %08X\n", fi.f, fi.i);
}

// prints:
// 1.00 is 3F800000
```

- 4-byte union
- **i** reads/writes an **int**
- **f** reads/writes a **float**

Endianness not noticeable:
members have same size.

Buffer "overrun"/"overflow" attacks

EXPLOITS VIA THE STACK AND THEIR PREVENTION

Arrays Bounds: Java, Python, C

```
class Bounds {
    public static void main(String[] args) {
        int[] x = new int[10];
        for (int i = 0; i <= x.length; i++) {
            x[i] = i;
        }
    }
}
```

```
$ javac Bounds.java
$ java Bounds
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 10
    at Bounds.main(Bounds.java:7)
```

```
x = [0] * 10

# not pythonic! but still...
for i in range(len(x) + 1):
    x[i] = i
```

```
$ python3 bounds.py
Traceback (most recent call last):
  File "bounds.py", line 5, in <module>
    x[i] = i
IndexError: list assignment index out of range
```

```
#include <stdio.h>
int main() {
    int x[10];
    for (int i = 0; i <= 10; i++) {
        x[i] = i;
    }
}
```

```
$ gcc bounds.c -o bounds
$ ./bounds
$
```

No failure! Why?

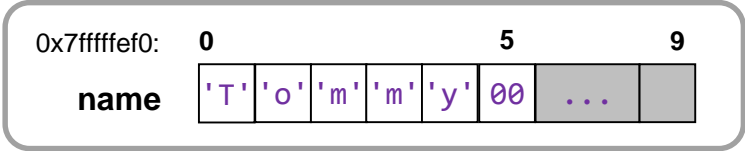
Arrays and Bounds Check

CS:APP 3.10.3

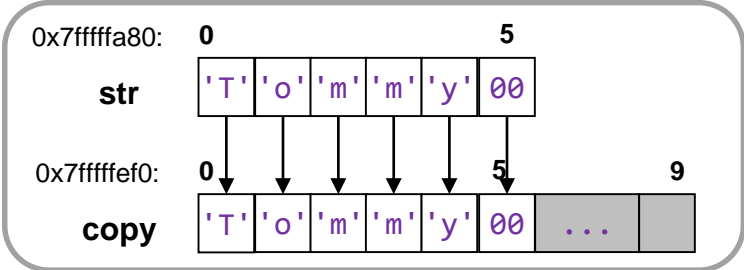
- Many functions, especially those related to strings, may not check the _____ of an array
- User or other input may _____ **a fixed size array**
 - Suppose the user types or passes "Tommy" to greet() or func1()
 - Note: **gets()** receives input from 'stdin' until the user enters '\n' and places the string in the given array (no bound checks!)

```
void greet() {
    char name[10];
    gets(name);
    ...
}
```

```
void func1(char *str) {
    char copy[10];
    strcpy(copy, str);
    ...
}
```



"Tommy" = 54 6f 6d 6d 79 00

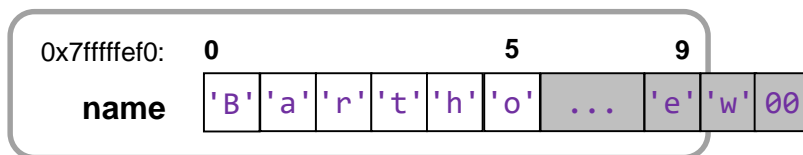


Arrays and Bounds Check

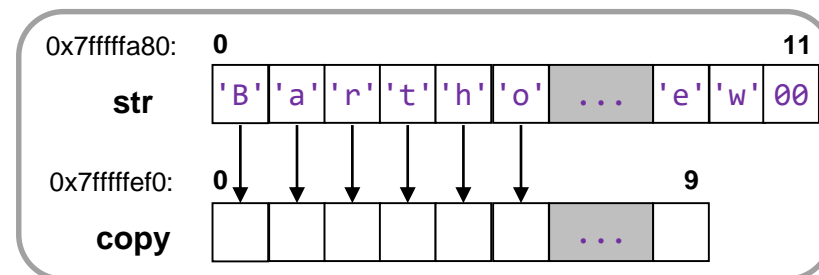
- Many functions, especially those related to strings, may not check the bounds of an array
- User or other input may overflow a fixed size array
 - Suppose the user types or passes "Tommy" to greet() or func1()
 - **Now suppose the user types or passes "Bartholomew"**

```
void greet() {
    char name[10];
    gets(name);
    ...
}
```

```
void func1(char *str) {
    char copy[10];
    strcpy(copy, str);
    ...
}
```



What are we overwriting?



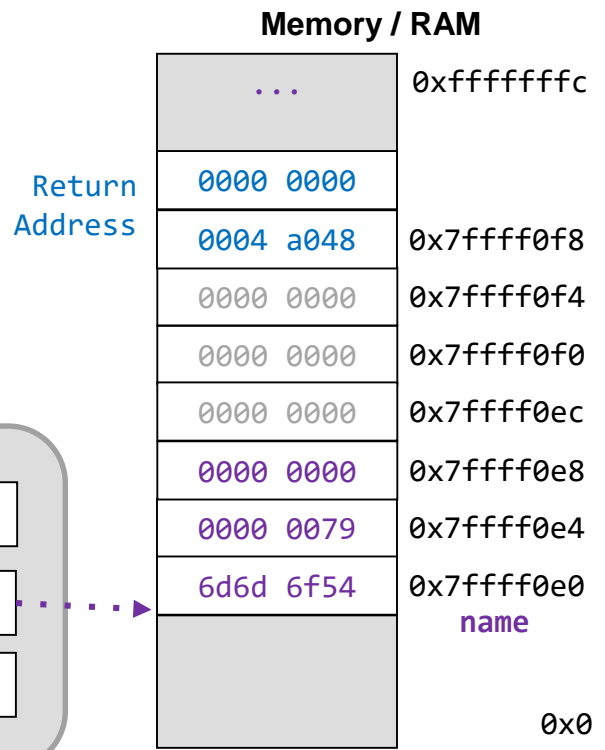
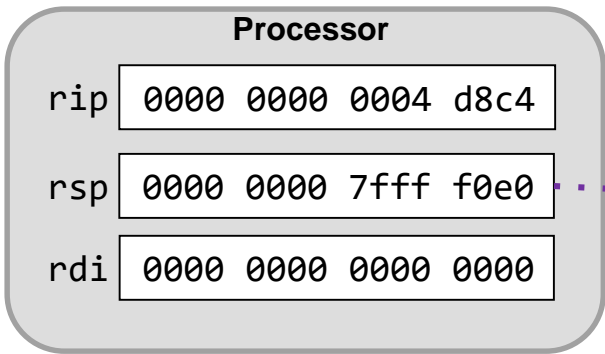
Buffer Overflow

- Now recall these **local arrays are stored on the _____** where the _____ **is also stored**
- gets() will copy as much as the user types (until they enter the '\n' = 0x0a), overwriting anything on the stack

```
void greet() {
    char name[12];
    gets(name);
    printf("Hello %s\n", name);
}
```

"Tommy" = 54 6f 6d 6d 79 00

```
greet:
    subq    $24, %rsp
    movq   %rsp, %rdi
    movl   $0, %eax
    call   gets
    movl   $.LC0, %esi
    movl   $1, %edi
    movl   $0, %eax
    call   __printf_chk
    addq   $24, %rsp
    ret
```



Overwriting the Return Address

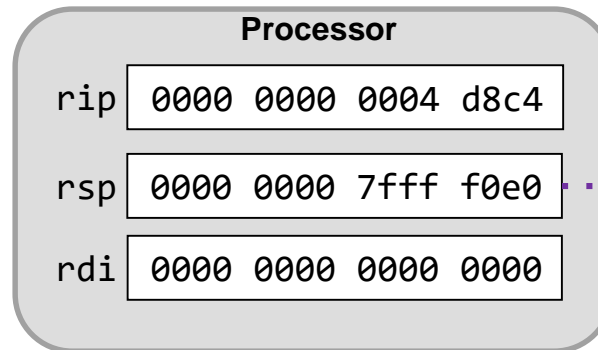
- An intelligent user could carefully craft a "long" input array and overwrite the return address with a desired value
- How could this be exploited?

```
void greet() {
    char name[12];
    gets(name);
    printf("Hello %s\n", name);
}
```

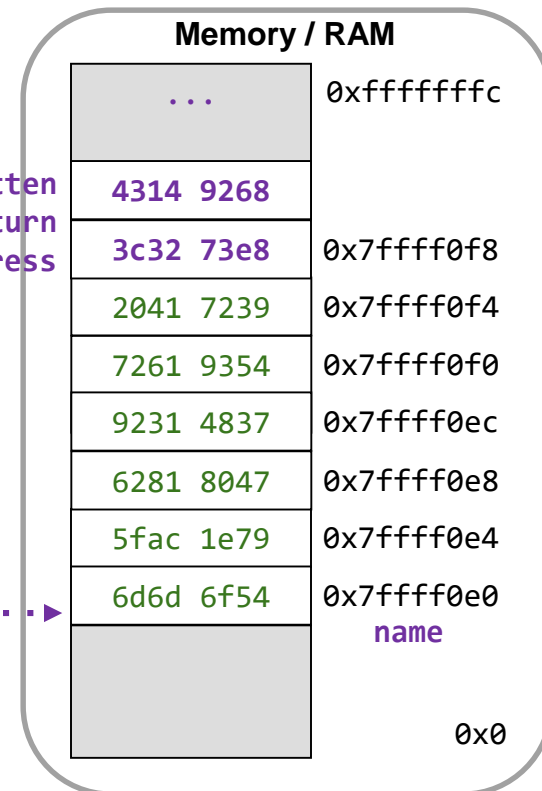
User string:

```
54 6f 6d 6d 79 1e ac 5f 47 80 81
62 37 48 31 92 54 93 61 72 39 72
41 20 e8 73 32 3c 68 92 14 43
```

```
greet:
    subq    $24, %rsp
    movq   %rsp, %rdi
    movl   $0, %eax
    call   gets
    movl   $.LC0, %esi
    movl   $1, %edi
    movl   $0, %eax
    call   __printf_chk
    addq   $24, %rsp
    ret
```



Overwritten
Return
Address



Executing Code

CS:APP 3.10.4

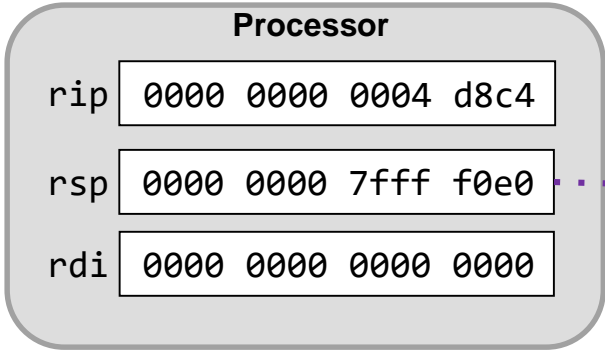
- We could determine the desired _____ for some sequence we want to execute on the machine and enter that as our string
- We can then craft a _____ to go to the starting location of our code

```
void greet() {
    char name[12];
    gets(name);
    printf("Hello %s\n", name);
}
```

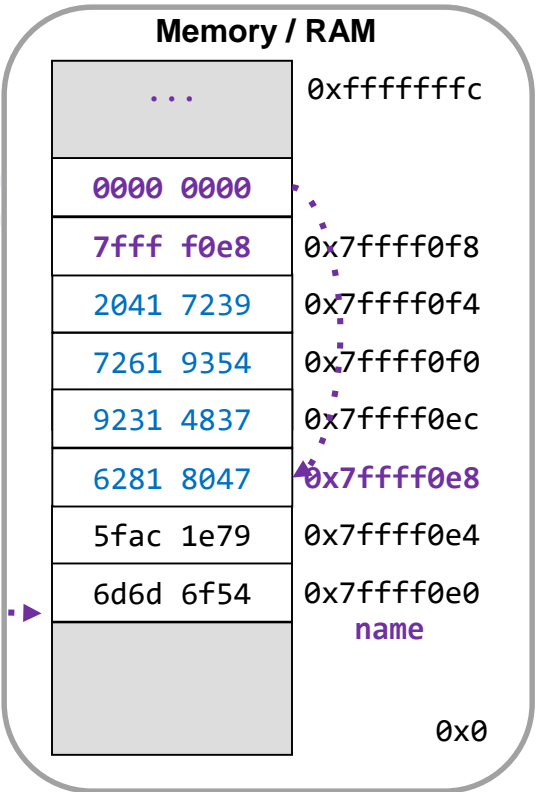
User string:

54	6f	6d	6d	79	1e	ac	5f	47	80	81
62	37	48	31	92	54	93	61	72	39	72
41	20	e8	f0	ff	7f	00	00	00	00	00

```
greet:
    subq    $24, %rsp
    movq   %rsp, %rdi
    movl   $0, %eax
    call   gets
    movl   $.LC0, %esi
    movl   $1, %edi
    movl   $0, %eax
    call   __printf_chk
    addq   $24, %rsp
    ret
```



Overwritten
Return
Address



Exploits

- Common code that we try to inject on the stack would start a _____ so that we can now type any other commands
- We can enter specific binary codes when a program prompts for a string by entering it in hex using the `\x` prefix



Typing: `"\x54\x6f\x5d..."` allows you enter the hex representation as a string

Methods of Prevention

- Various methods have been devised to prevent or make it harder to exploit this code
 - **Better _____ that do not allow an overrun**
`strcpy (char* dest, char* src)`
`strncpy(char* dest, char* src, size_t len)`
 - Add a stack protector (e.g., _____)
 - Address space layout _____ (ASLR) techniques
 - _____ control bits

Canary Values

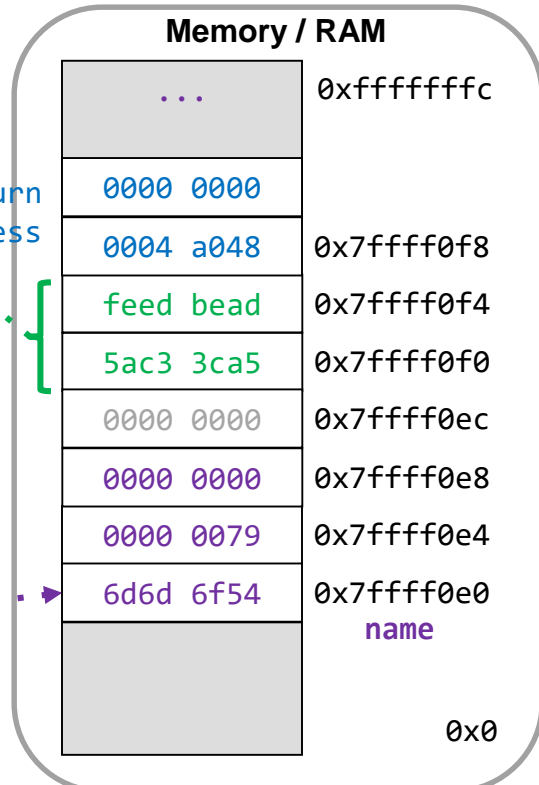
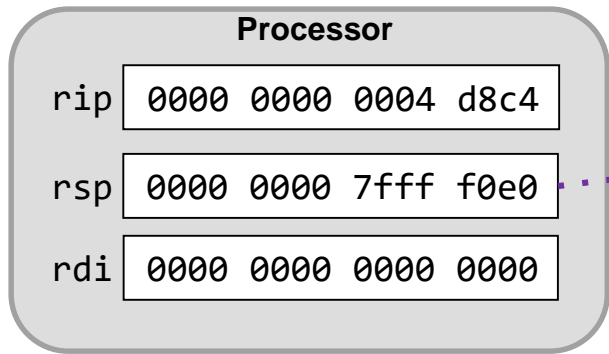
- Compiler will insert code to generate and store a _____ value between the return address and the _____
- Before returning it will _____ whether this value has been _____ (by a buffer overflow) and raise an error if it has

```
greet:
    subq    $24, %rsp
    movq    %fs:40, %rax
    movq    %rax, 16(%rsp)
    movq    %rsp, %rdi
    movl    $0, %eax
    call    gets
    movl    $.LC0, %esi
    movl    $1, %edi
    movl    $0, %eax
    call    __printf_chk
    movq    16(%rsp), %rax
    xorq    %fs:40, %rax
    je      .L2
    call    __stack_chk_fail

.L2:
    addq    $24, %rsp
    ret
```



This Photo by Unknown Author is licensed under [CC BY-NC](#)



Return Address

feed bead

6d6d 6f54

Address Space Layout Randomisation

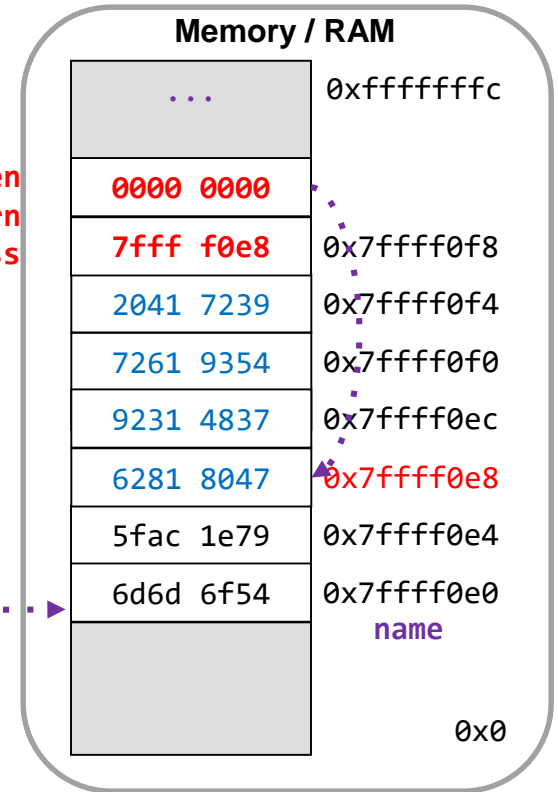
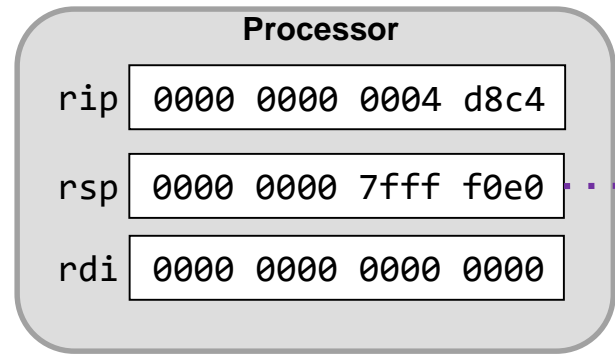
- Notice that to call our exploit code we have to know the _____ address on the stack where our exploit code starts (e.g. 0x7ffff0e8) and make that our RA
- The stack usually starts at the _____ address when each program runs so it might be fairly easy to predict
 - Run the program on our own server to learn its behavior, then run on a server we want to exploit
- Idea: Randomize where the stack will _____

```
void greet()
{
    char name[12];
    gets(name);
    printf("Hello %s\n");
}
```

User string:
 54 6f 6d 6d 79 1e ac 5f 47 80 81
 62 37 48 31 92 54 93 61 72 39 72
 41 20 e8 f0 ff 7f 00 00 00 00

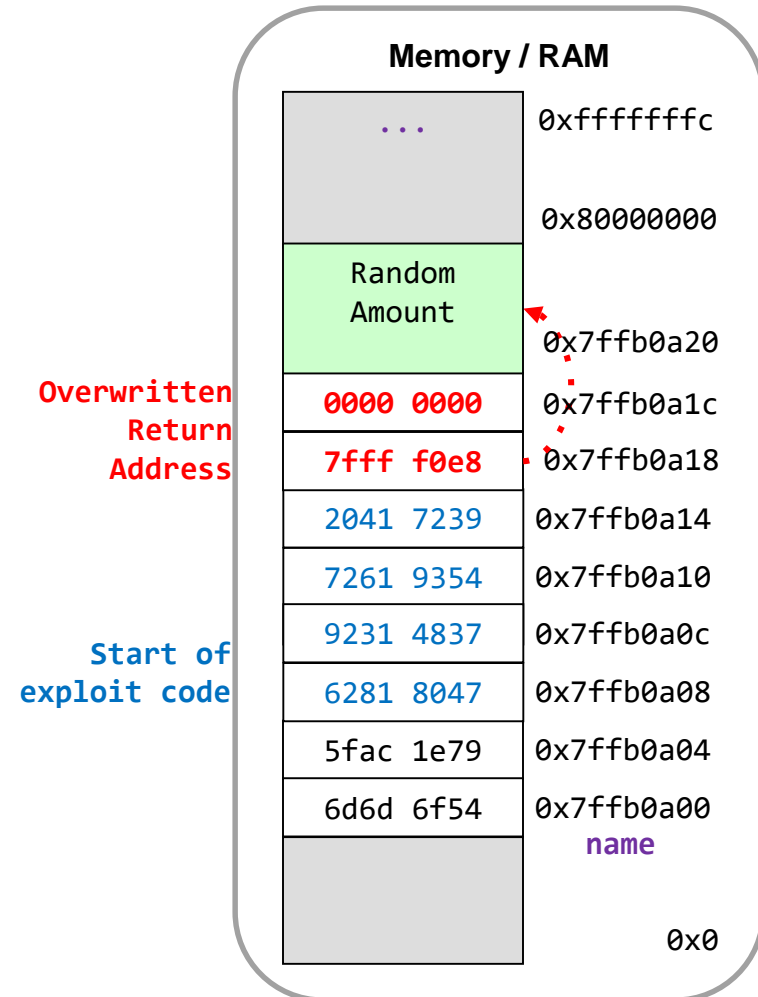
Overwritten
Return
Address

```
greet:
    subq    $24, %rsp
    movq    %rsp, %rdi
    movl    $0, %eax
    call    gets
    movl    $.LC0, %esi
    movl    $1, %edi
    movl    $0, %eax
    call    __printf_chk
    addq    $24, %rsp
    ret
```



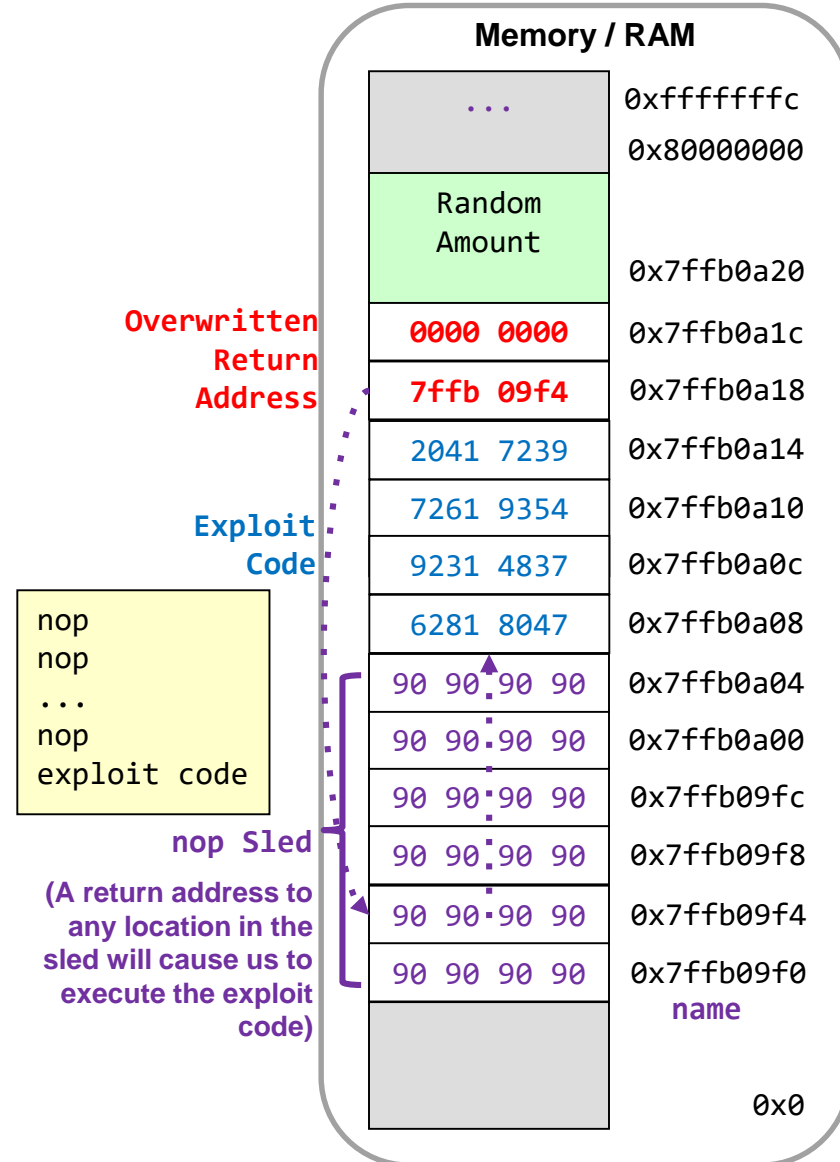
How the OS randomizes the layout

- The OS can allocate a random amount of space on the stack each time a program is executed to make it harder for an attacker to succeed in an exploit
 - This is referred to as **ASLR (Address Space Layout Randomization)**
- Our previous exploit string would now have a return address that does not lead to our exploit code and likely result in a _____ rather than execution of the exploit code



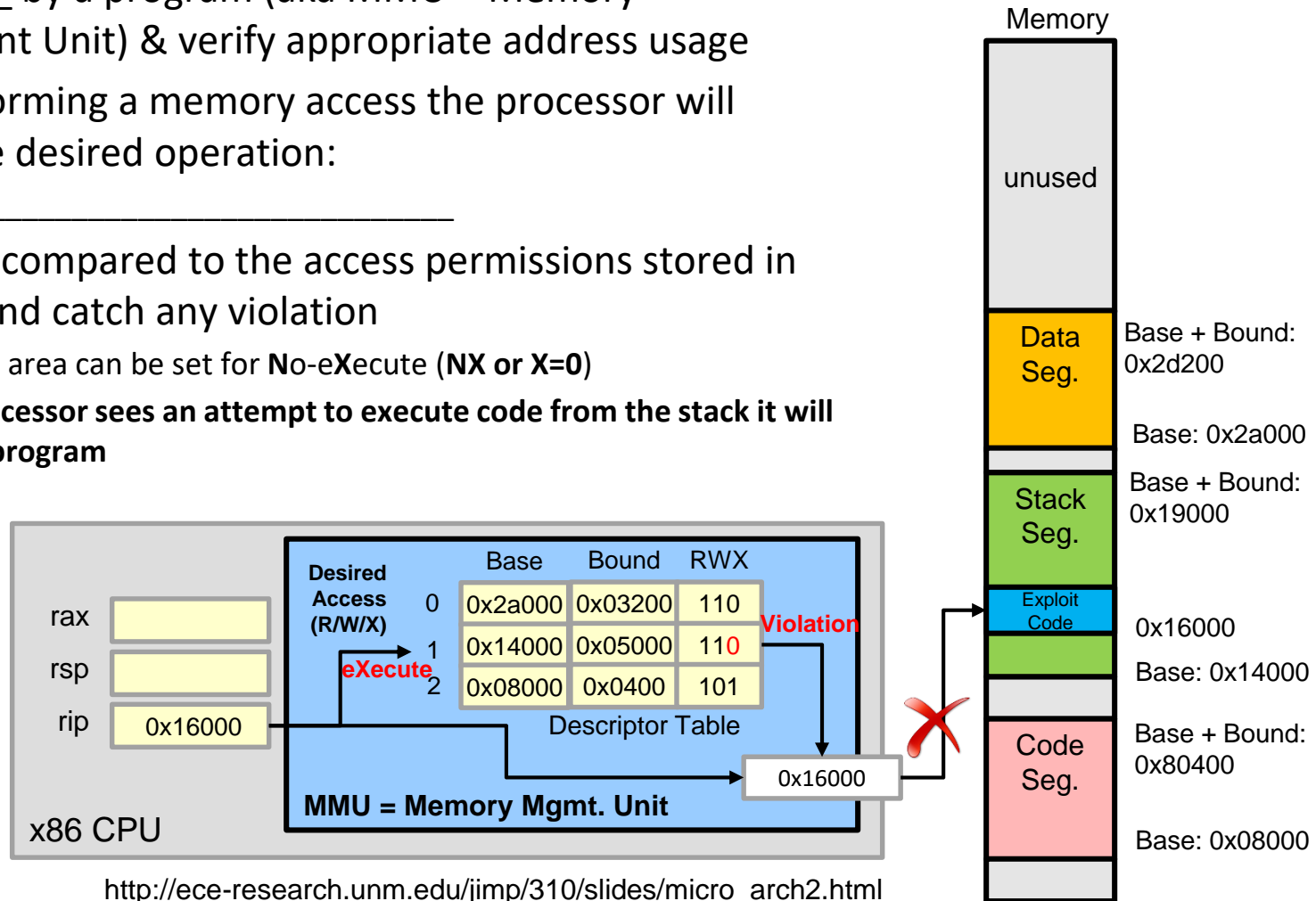
nop sleds

- **Fact:** Most instruction sets have a 'nop' instruction that is an instruction that does nothing
 - Can also just use an instruction that does very little (e.g. `movq %rsp, %rsp`)
- **Idea:** _____ as many 'nop' instructions as possible in the buffer before the exploit code
- **Effect:** Now our guess for the RA does not need to be _____ but **anywhere in the _____ of nops**
 - This yields a higher chance of actually landing in a location that will eventually cause the exploit to be executed



Memory Protection & Permissions

- Processors have hardware to help track areas of memory _____ by a program (aka MMU = Memory Management Unit) & verify appropriate address usage
- When performing a memory access the processor will indicate the desired operation:
 - _____
- This will be compared to the access permissions stored in the MMU and catch any violation
 - The stack area can be set for **No-eXecute (NX or X=0)**
 - If the processor sees an attempt to execute code from the stack it will halt the program**



Code Injection Attacks

- These buffer overflow exploits have all tried to copy code into some area of memory and then have it be executed
- We refer to this approach as **code-injection** attacks
- To try a code injection attack you need to disable these protections... check the discussion slides!

Run it at home

```
$ cat hello.c
#include <stdio.h>

void unreachable() {
    printf("Impossible.\n");
}

void hello() {
    char buffer[6];
    scanf("%s", buffer);
    printf("Hello, %s!\n", buffer);
}

int main() {
    hello();
    return 0;
}

$ gcc -no-pie hello.c -o hello

$ objdump -d hello | grep unreachable
0000000000401142 <unreachable>:

$ objdump -d hello | grep -A8 '<hello>:'
0000000000401155 <hello>:
 401155:    55                push   %rbp
 401156:    48 89 e5          mov    %rsp,%rbp
 401159:    48 83 ec 10       sub   $0x10,%rsp
 40115d:    48 8d 45 fa       lea   -0x6(%rbp),%rax
 401161:    48 89 c6          mov    %rax,%rsi
 401164:    48 8d 3d a5 0e 00 00 lea   0xea5(%rip),%rdi    # 402010 <_IO_stdin_used+0x10>
 40116b:    b8 00 00 00 00   mov    $0x0,%eax
 401170:    e8 db fe ff ff   callq 401050 <__isoc99_scanf@plt>

$ (echo -n 'World!'; echo '11223344556677884211400000000000' | xxd -r -p) | ./hello
Hello, World!"3DUfw0B@!
Impossible.
```

Return Oriented Programming

- What if the stack is marked as non-executable? And its position randomized?
- We can use **return-oriented programming**
- Key idea: find the attack instructions inside of those that already exist in the code segment

Return Oriented Programming

What if the program is more secure?

- It uses randomization to avoid fixed stack positions.
- The stack is marked as non-executable.

Idea: return-oriented programming

- Find **gadgets** in executable areas.
- Gadget: short sequence of instructions followed by **ret** (0xc3)

Often, it is possible to find useful instructions within the byte encoding of other instructions.

```
void setval_210(unsigned *p) {  
    *p = 3347663060U;  
}
```

```
000000000400f15 <setval_210>:  
400f15: c7 07 d4 48 89 c7    movl $0xc78948d4, (%rdi)  
400f1b: c3                   retq
```

48 89 c7 encodes the x86_64 instruction **movq %rax, %rdi**

To start this gadget, set a return address to 0x400f18 (use little-endian format)

Finding the right instruction

Operation		Register <i>R</i>			
		%al	%cl	%dl	%bl
andb	<i>R, R</i>	20 c0	20 c9	20 d2	20 db
orb	<i>R, R</i>	08 c0	08 c9	08 d2	08 db
cmpb	<i>R, R</i>	38 c0	38 c9	38 d2	38 db
testb	<i>R, R</i>	84 c0	84 c9	84 d2	84 db

Operation		Register <i>R</i>							
		%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
popq	<i>R</i>	58	59	5a	5b	5c	5d	5e	5f

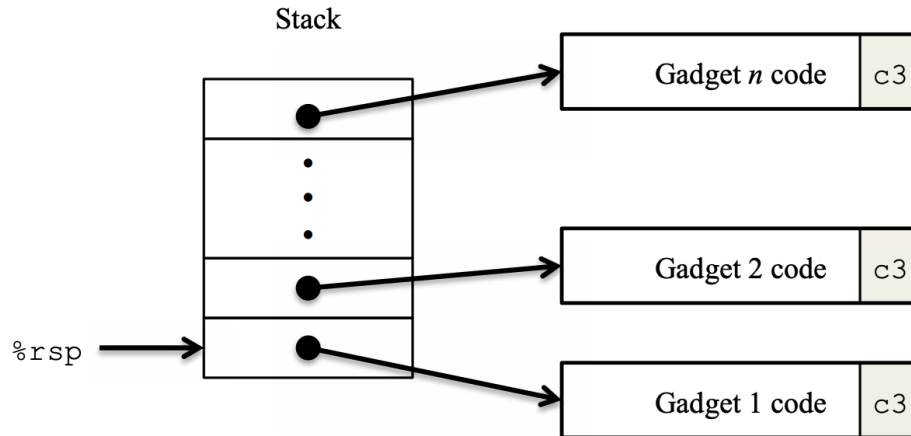
movl *S, D*

Source <i>S</i>	Destination <i>D</i>							
	%eax	%ecx	%edx	%ebx	%esp	%ebp	%esi	%edi
%eax	89 c0	89 c1	89 c2	89 c3	89 c4	89 c5	89 c6	89 c7
%ecx	89 c8	89 c9	89 ca	89 cb	89 cc	89 cd	89 ce	89 cf
%edx	89 d0	89 d1	89 d2	89 d3	89 d4	89 d5	89 d6	89 d7
%ebx	89 d8	89 d9	89 da	89 db	89 dc	89 dd	89 de	89 df
%esp	89 e0	89 e1	89 e2	89 e3	89 e4	89 e5	89 e6	89 e7
%ebp	89 e8	89 e9	89 ea	89 eb	89 ec	89 ed	89 ee	89 ef
%esi	89 f0	89 f1	89 f2	89 f3	89 f4	89 f5	89 f6	89 f7
%edi	89 f8	89 f9	89 fa	89 fb	89 fc	89 fd	89 fe	89 ff

movq *S, D*

Source <i>S</i>	Destination <i>D</i>								
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi	
%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7	
%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf	
%rdx	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7	
%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df	
%rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7	
%rbp	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef	
%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7	
%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff	

Using multiple gadgets



- The stack contains a sequence of gadget addresses.
- Each gadget consists of a series of instruction bytes, with the final one being 0xc3 (encoding the ret instruction).
- When the program executes a ret instruction starting with this configuration, it will initiate a chain of gadget executions, with the ret instruction at the end of each gadget causing the program to jump to the beginning of the next.

Purpose of %rbp as "Base" or "Frame" Pointer

STACK FRAMES

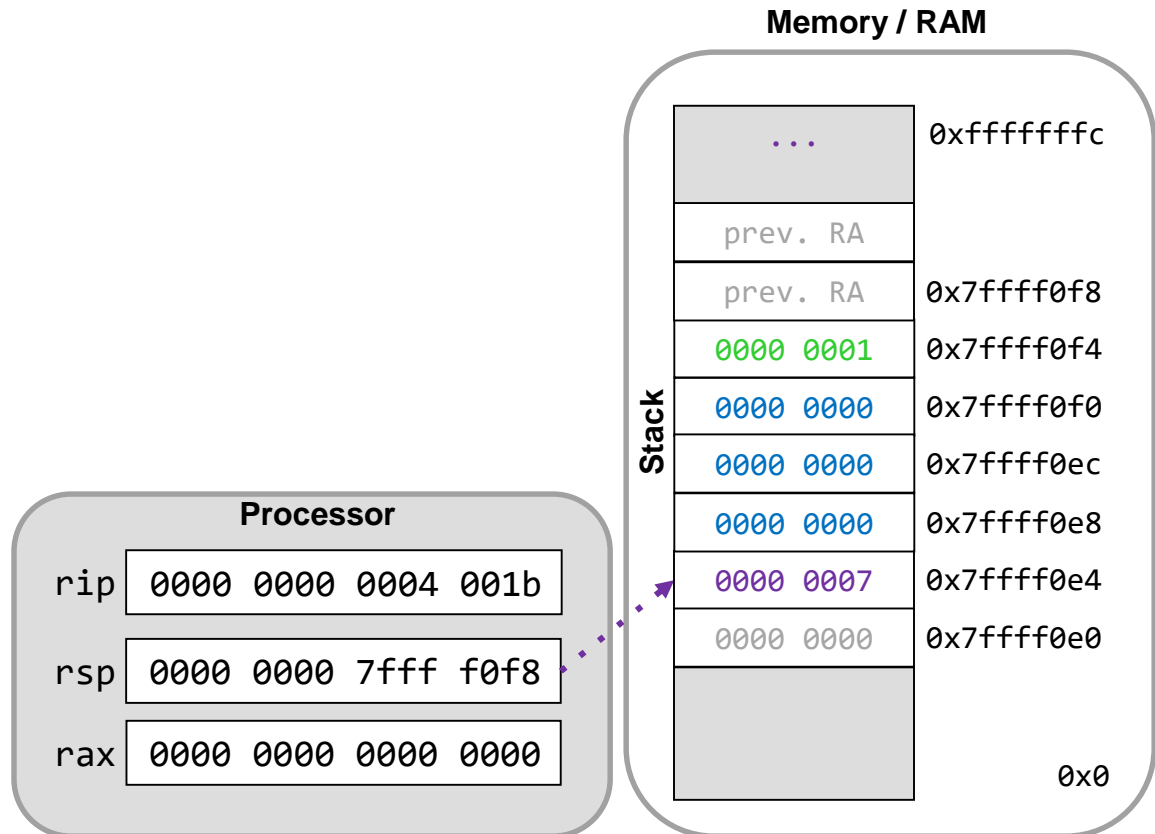
Stack Frame Motivation 1

CS:APP 3.10.5

- Under certain circumstances the compiler cannot easily generate code using the stack pointer (%rsp) alone
 - The most common of these cases is when the allocation size is variable

```
int varArray(int n) Compiler doesn't know n
{ when it generates the code
    int temp1=7, data[n], temp2=1;
    ...
}
```

```
movl (%rsp), %eax # access temp1
movl 4(%rsp), %ecx # access data[0]
movl ??(%rsp), %edx # access temp2?
```

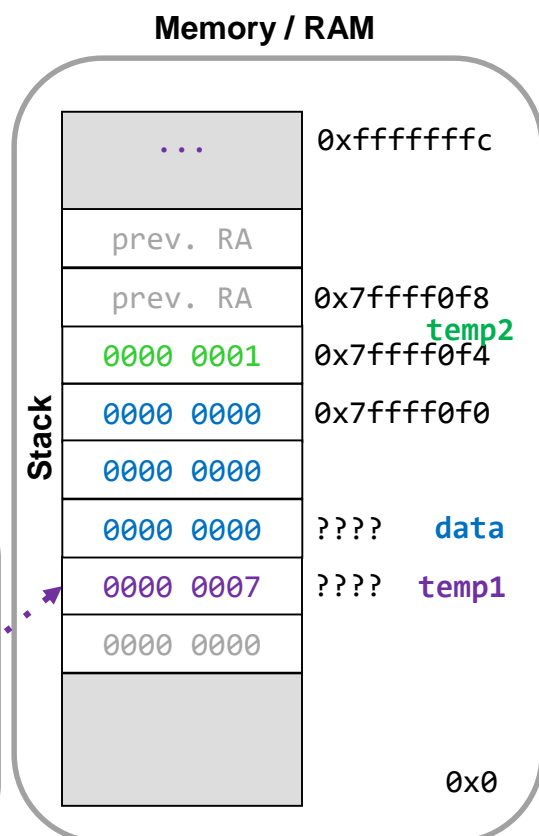
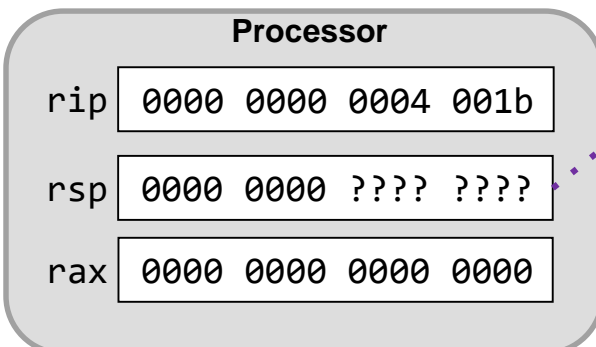


Stack Frame Motivation 2

- We access local variables using a constant displacement from the %rsp (i.e. 8(%rsp))
- But if we have to move the stack pointer up a *variable* amount (only known at runtime) there is **no constant displacement** the compiler can use to access some local variables (e.g. temp2)
 - Would need to compute the offset based on the variable size and use (reg1,reg2,s) style address mode which would be slower

```
int varArray(int n)
{
    int temp1=7, data[n], temp2=1;
    ...
}
```

```
movl (%rsp), %eax # access temp1
movl 4(%rsp), %ecx # access data[0]
movl ??(%rsp), %edx # access temp2?
```



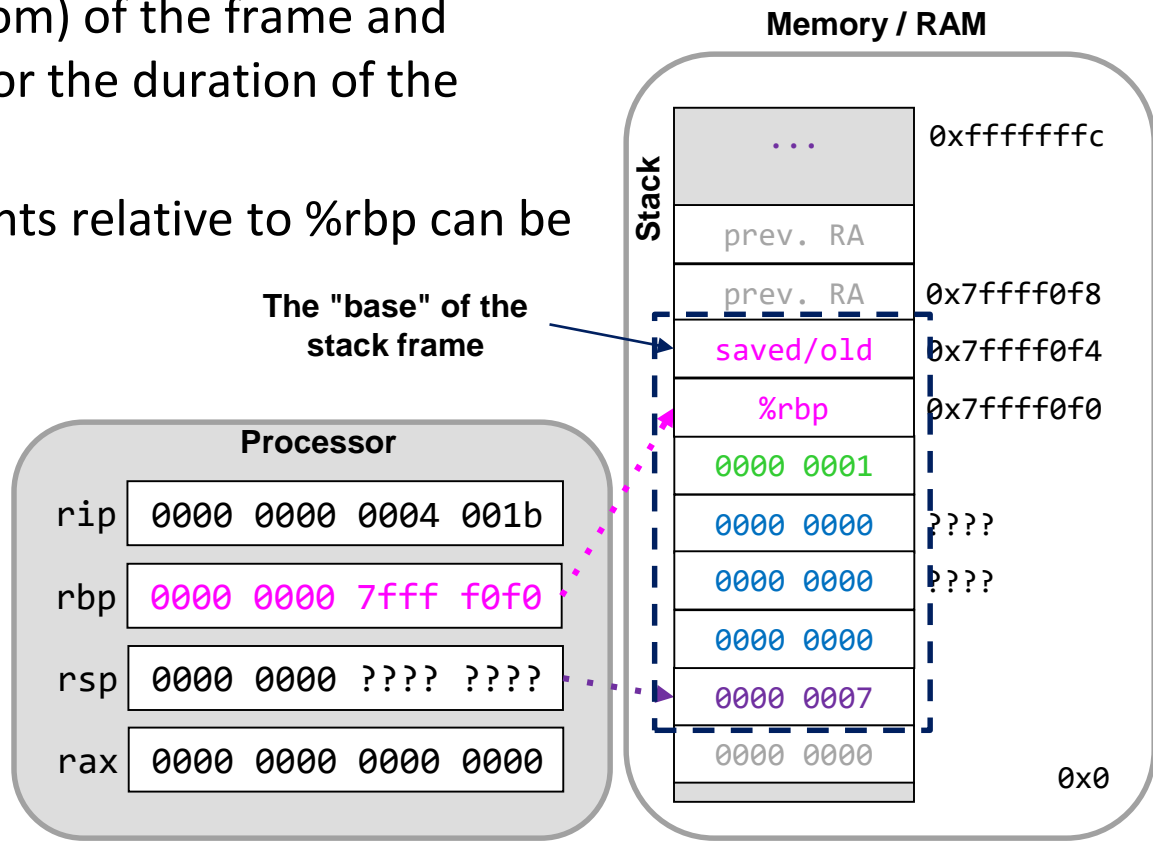
Base/Frame Pointer

- Since we may not know the offsets of variables relative to the stack pointer, a common solution is to use a second register call the **base or frame pointer**
 - **x86 uses %rbp for this purpose**
- It points at the base (bottom) of the frame and remains stable/constant for the duration of the procedure
- Now constant displacements relative to %rbp can be used by the compiler

Main point: The base/frame pointer will always point to a **known, stable location** and other variables will be at constant offsets from that location

```
int varArray(int n)
{
    int temp1=7, data[n], temp2=1;
    ...
}
```

```
movl (%rsp), %eax # access temp1
movl 4(%rsp), %ecx # access data[0]
movl -4(%rbp), %edx # access temp2
```



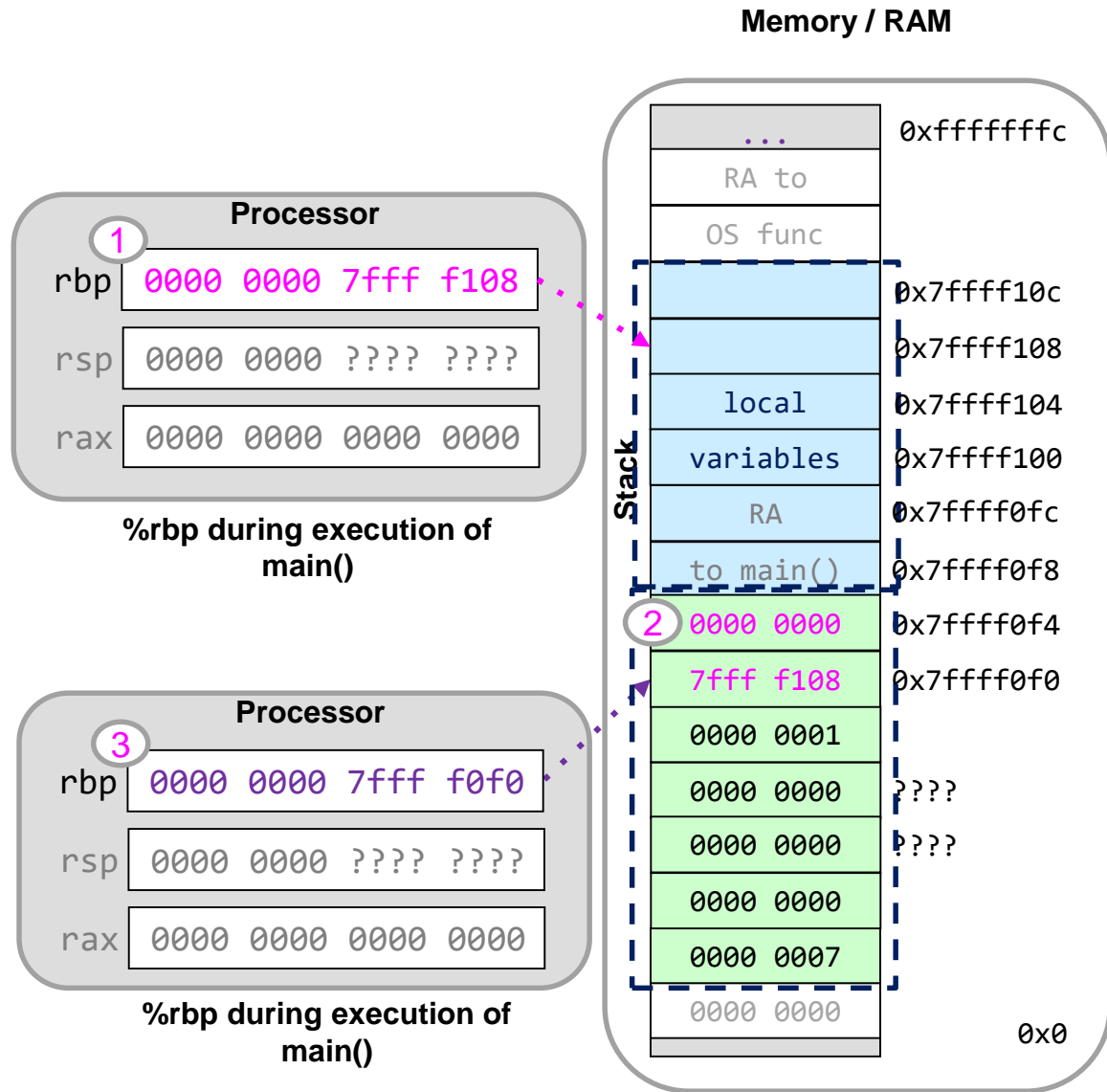
Saving the Old Base Pointer

- Since each function call needs its own value for %rbp we must save/restore it each time we call a new function
- Generally we setup the base pointer as the first task when starting a new function

```

int main()
{
    int num;
    ...
    varArray(num)
}

int varArray(int n)
{
    int temp1=7, data[n], temp2=1;
    ...
}
```



Setting up the Base Pointer

- Below is the common **preamble** for a function as it saves the old base pointer and sets up its own
- The base pointer can be used during execution
- The last 3 instructions are the **postamble** to restore the old base pointer and then exit

```

1  varArray:
    pushq %rbp      # Save main's %rbp
    movq  %rsp, %rbp # Set up new %rbp
    subq  $16, %rsp  # Allocate some space
    ...
2  movl  -8(%rbp), %edx # access temp2
    ...
    movq  %rbp, %rsp # Deallocate stack space
    popq  %rbp      # Restore main's %rbp
    ret
    
```

