CS356 Unit 7

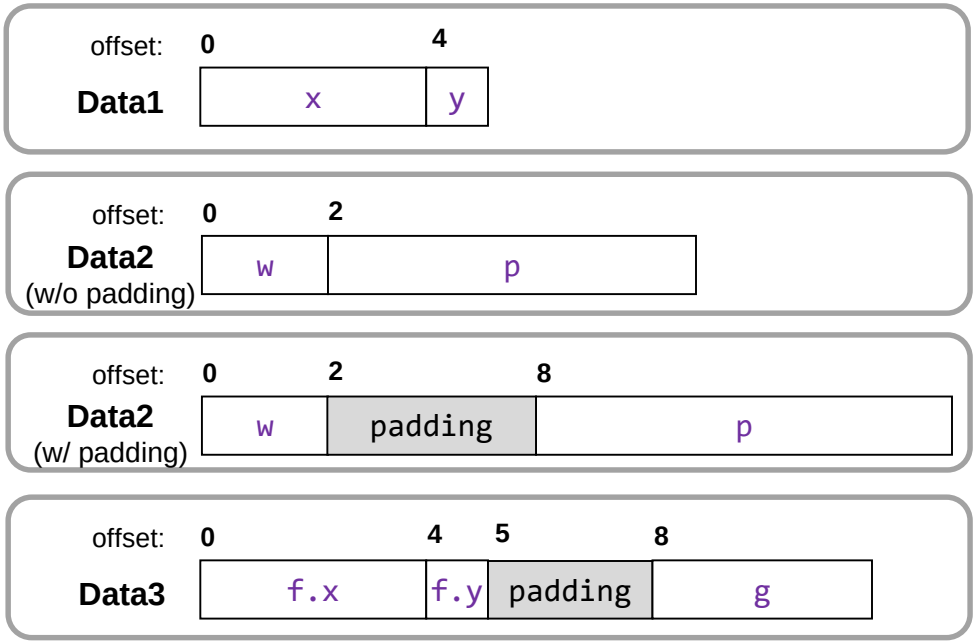
Data Layout &
Intermediate Stack Frames

Structs

CS:APP 3.9.1

- Structs (classes) are just collections of heterogeneous data
- Each member is usually laid out in consecutive memory locations (with some padding inserted to ensure alignment)
  - Padding is necessary if there are alignment requirements for certain data types
  - Intel machines don't require alignment but perform better when it is used

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

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

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



Class Example (1)

- Classes are just structs and lay out their data the same way
- Member function calls are converted to normal function calls with the '`this`' pointer passed as the first argument

```
class ObjA {
 public:
  char c;  int x;
  void update(int z);
};

void ObjA::update(int z)
{  c = 'a'; x = z; }

int main()
{
  ObjA item;
  item.update(7);
  cout << item.c << item.x << endl;
  return 0;
}
```

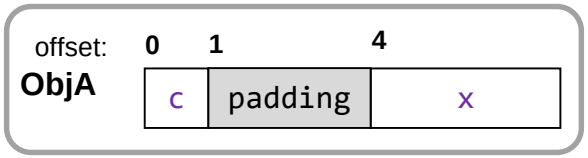
What you write…

```
class ObjA {
 public:
  char c;  int x;
  void update(int z);
};

void ObjA::update(ObjA* this, int z)
{  this->c = 'a'; this->x = z; }

int main()
{
  ObjA item;
  update(&item, 7);
  cout << item.c << item.x << endl;
  return 0;
}
```

…what the compiler sees

Classes Example (2)

- Structs (classes) are just collections of heterogeneous data




```
class ObjA {
 public:
  char c;  int x;
  void update(int z);
};

void ObjA::update(ObjA* this, int z)
{  this->c = 'a'; this->x = z; }

int main()
{
  ObjA item;
  update(&item, 7);
  cout << item.c << item.x << endl;
  return 0;
}
```

```
_ZN4ObjA6updateEi:
    movb    $97, (%rdi)
    movl    %esi, 4(%rdi)
    ret

main:
    pushq   %rbx
    subq    $32, %rsp
    movq    %fs:40, %rax
    movq    %rax, 24(%rsp)
    xorl    %eax, %eax
    movl    $7, %esi
    leaq    16(%rsp), %rdi
    call    _ZN4ObjA6updateEi
    ...
```

%rdi

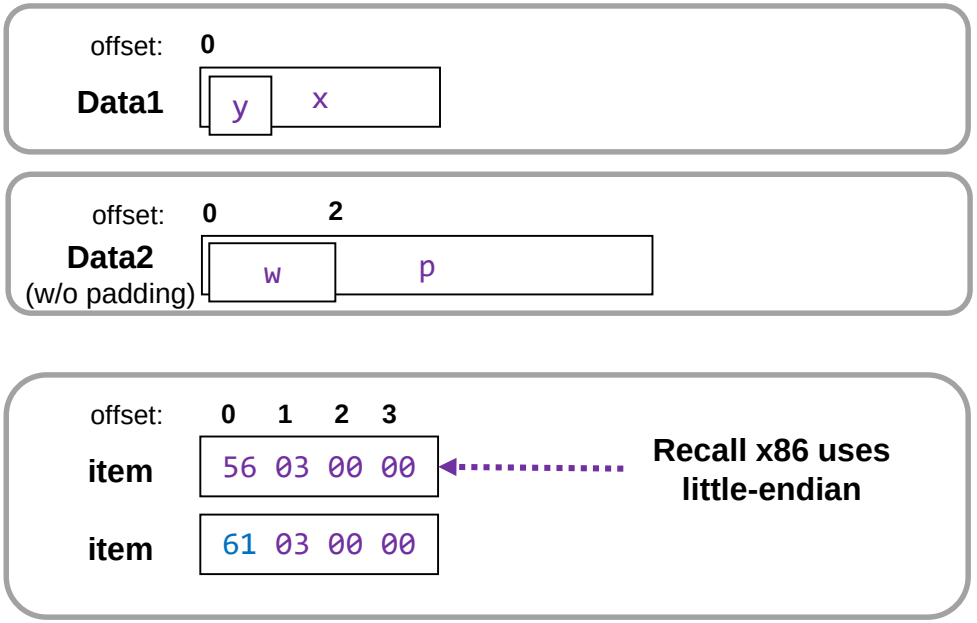
%esi

Unions

CS:APP 3.9.2

- Unions allow multiple variable types to occupy the same memory with the understanding only one type will be in use at any time for a given object
  - All elements start at offset 0
  - The size of the union is simply the size of the biggest 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';
}
```



Buffer "overrun"/"overflow" attacks

EXPLOITS VIA THE STACK AND THEIR PREVENTION

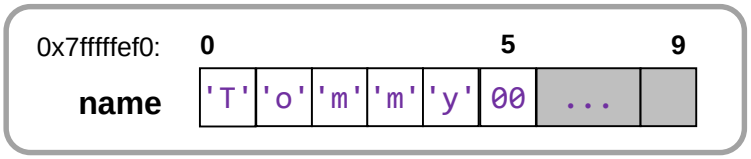
Arrays and Bounds Check

CS:APP 3.10.3

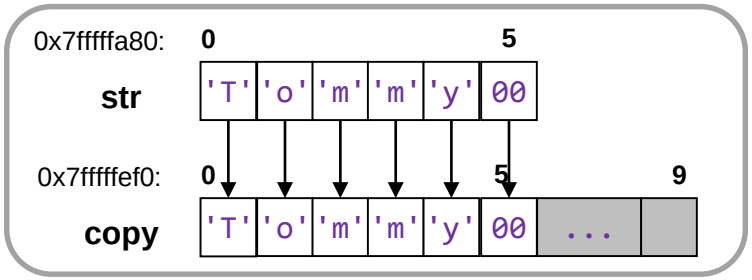
- Many functions, especially those related to strings, may not check the bounds of an array
- User or other input may overflow a fixed size array
  - Suppose the user types or passes "Tommy" to greet() or func1()
  - Note: gets() receives input from 'stdin' until the user hits 'Enter' and places the string in the given array

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

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



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

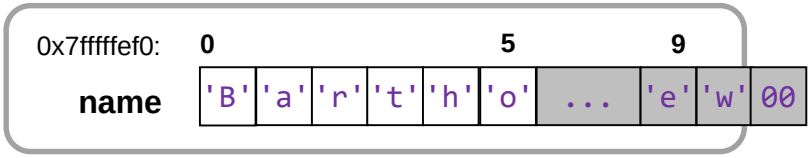


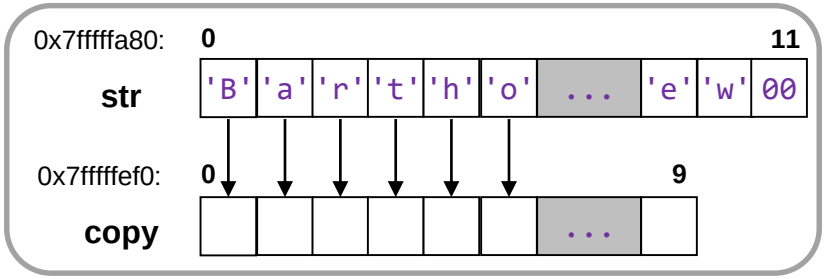
Arrays and Bounds Check

- 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); // cin >> name;
  ...
}
```

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





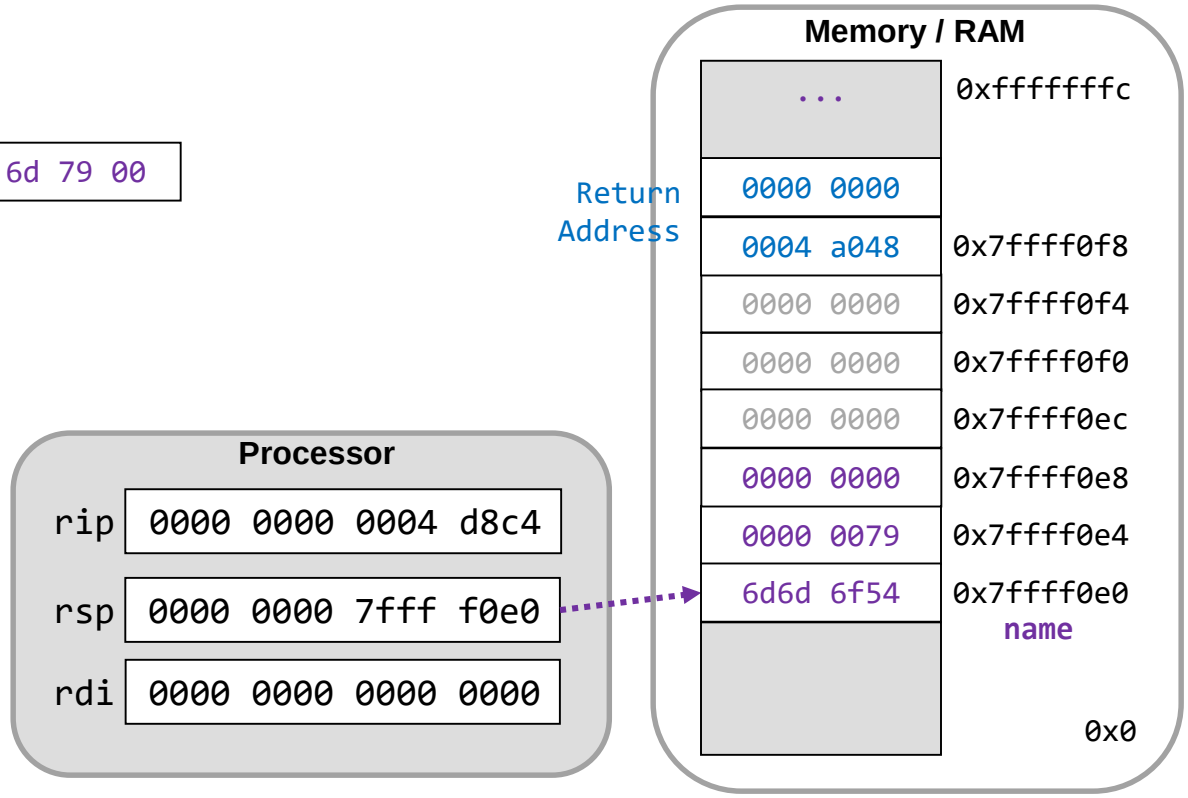
Buffer Overflow

- Now recall these local arrays are stored on the stack where other variables and the return address are stored
- In this example, 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");
}
```

"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
```



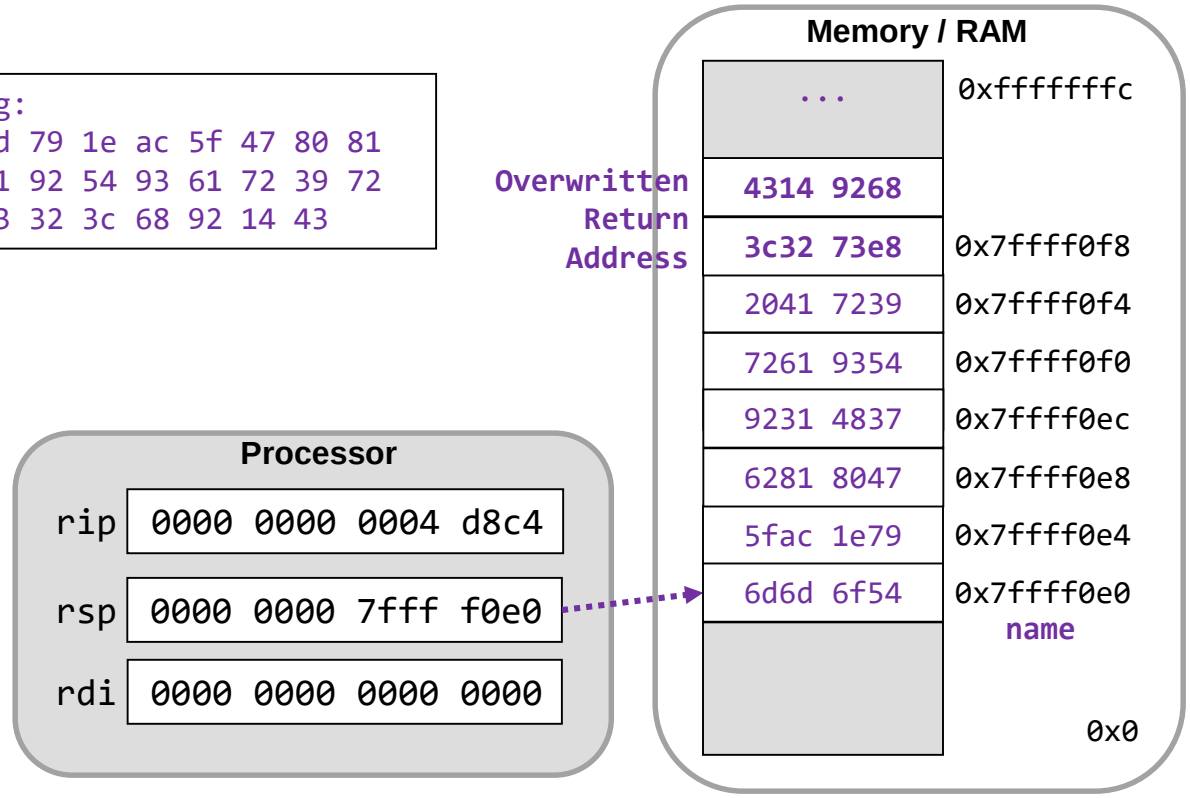
Overwriting the Return Address

- 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");
}
```

```
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
```



Executing Code

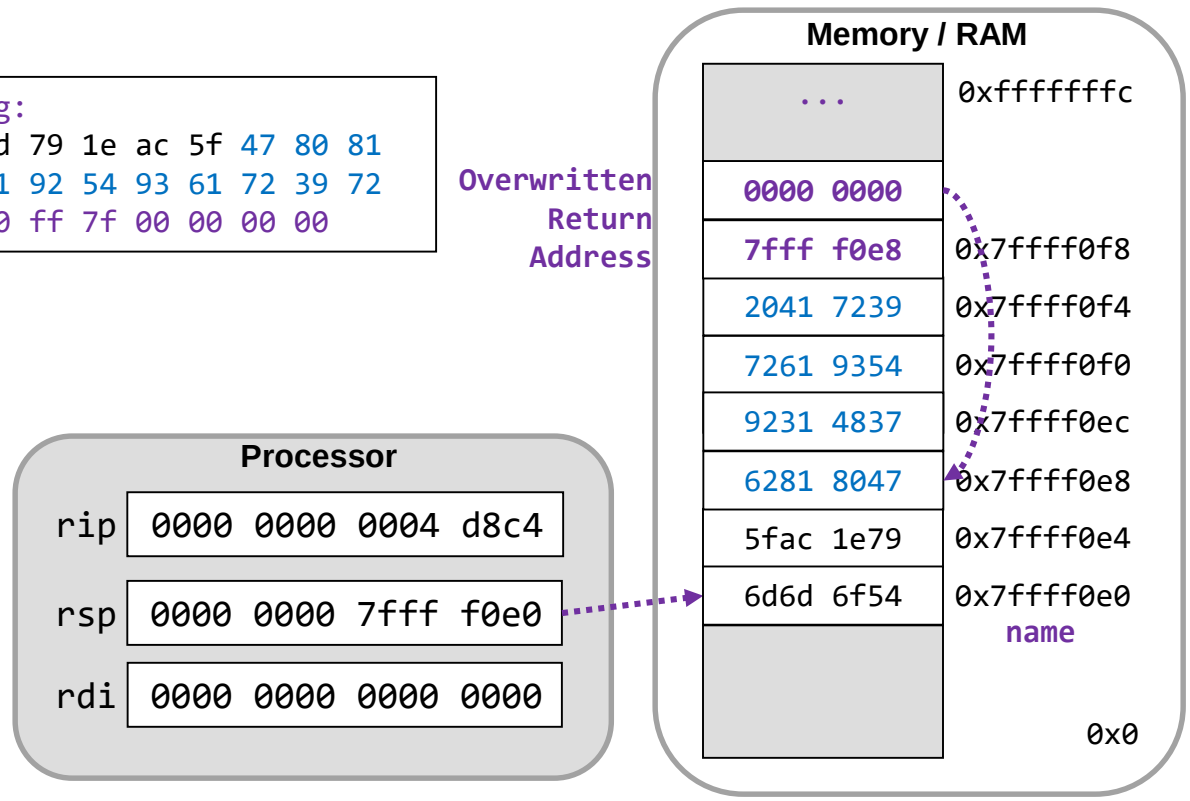
CS:APP 3.10.4

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

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

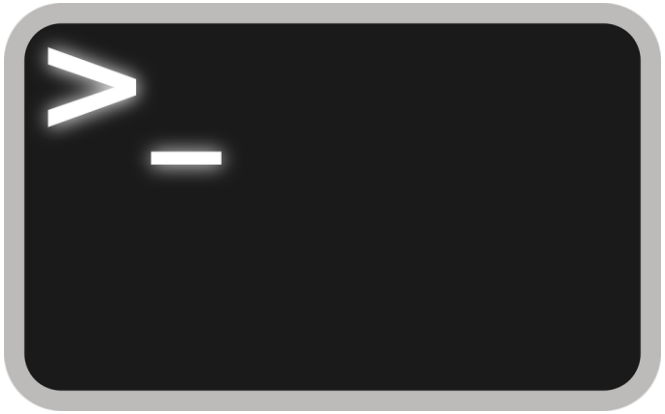
```
User string:
54 6f 6d 6d 79 1e ac 5f 47 80 81
62 37 48 31 92 54 93 61 72 39 72
41 20 e8 f0 ff 7f 00 00 00 00
```

```
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
```



Exploits

- Common code that we try to inject on the stack would start a shell 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 libraries that do not allow an overrun
    - `strcpy(char* dest, char* src) => strncpy(char* dest, char* src, size_t len)`
  - Add a stack protector (a.k.a. canary)
  - Address space layout randomization (ASLR) techniques
  - Privilege/access control bits

Canary Values

- Compiler will insert code to generate and store a unique value between the return address and the local variables
- Before returning it will check whether this value has been altered (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    24(%rsp), %rax
        xorq    %fs:40, %rax
        je      .L2
        call    __stack_chk_fail
.L2:
        addq    $24, %rsp
        ret
```

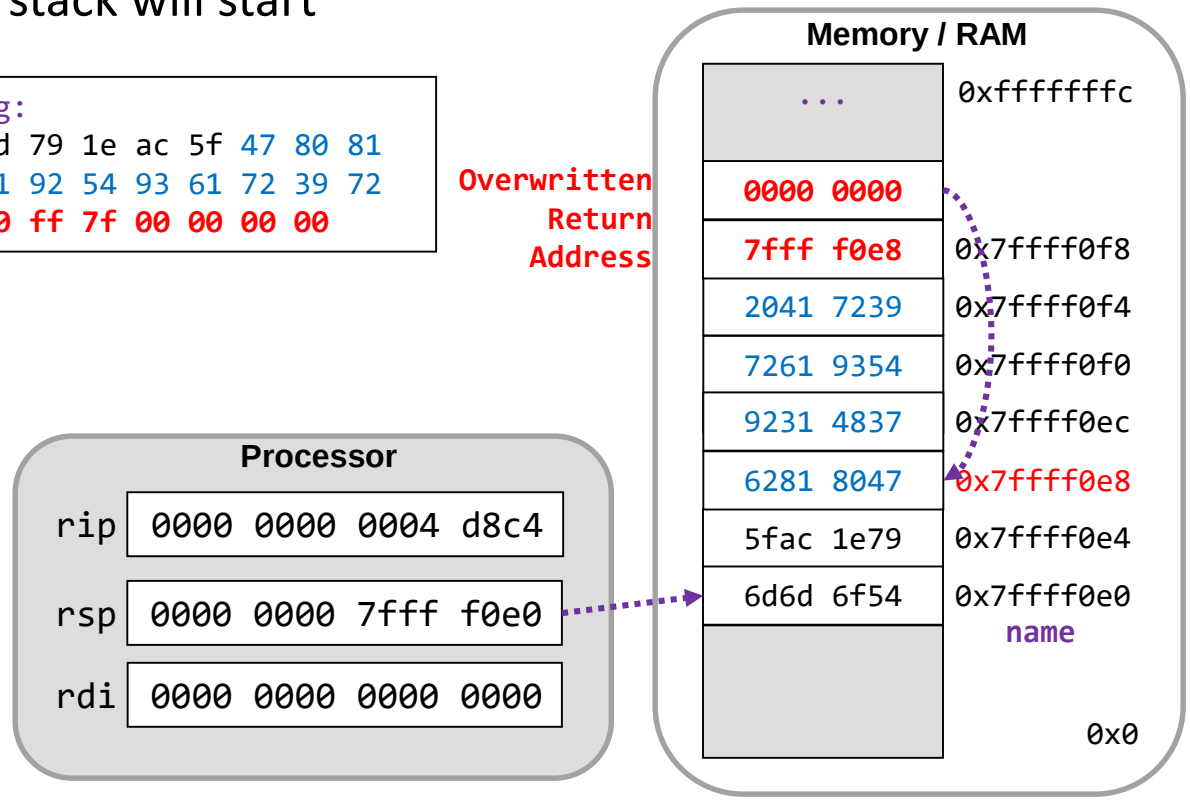
Motivation for Randomization

- Notice that to call our exploit code we have to know the exact address on the stack where our exploit code starts (e.g. 0x7ffff0e8) and make that our RA
- The stack usually starts at the same 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 start

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

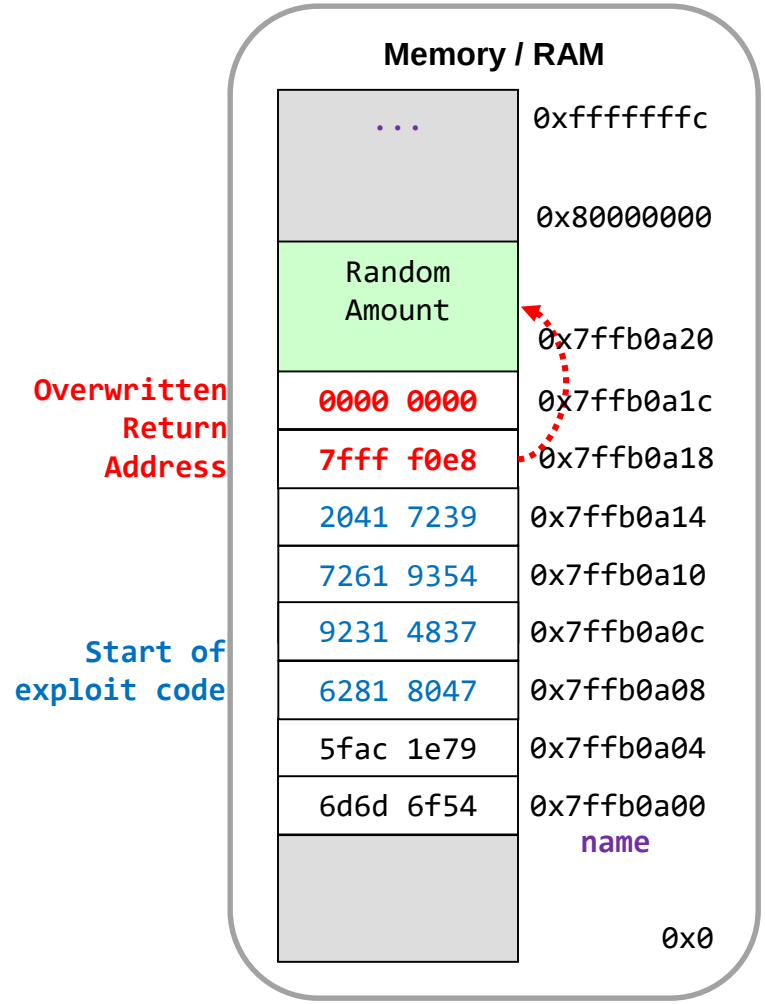
```
User string:
54 6f 6d 6d 79 1e ac 5f 47 80 81
62 37 48 31 92 54 93 61 72 39 72
41 20 e8 f0 ff 7f 00 00 00 00
```

```
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
```



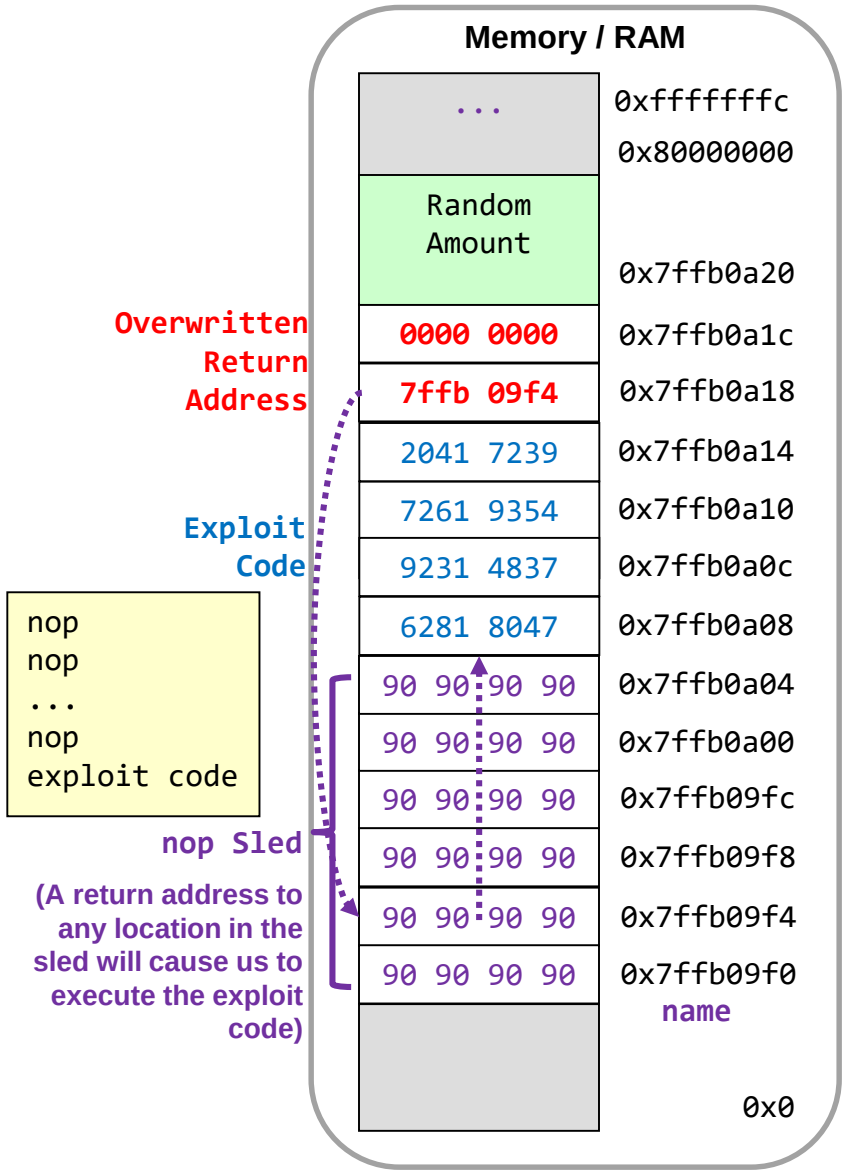
Motivation for Randomization

- 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)**
- Note: Our previous exploit string would now have a return address that does not lead to our exploit code and likely result in a crash rather than execution of the exploit code



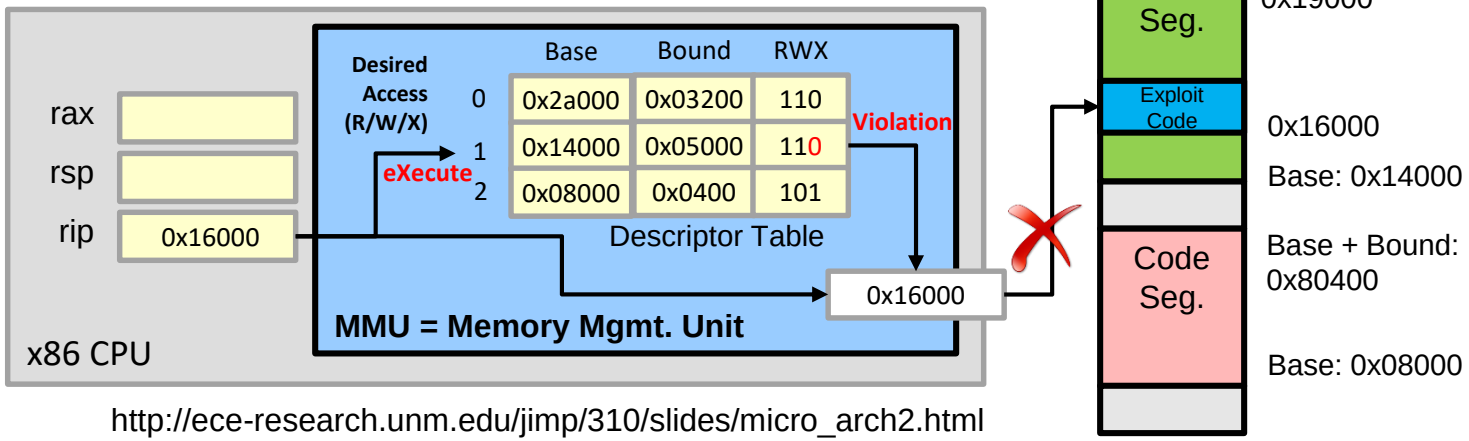
nop Sleds

- 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**: Prepend 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 exact but anywhere in the range 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

- Processor have hardware to help track areas of memory used by a program (aka MMU = Memory Management Unit) & verify appropriate address usage
- When performing a memory access the processor will indicate the desired operation:
  - Fetch (eXecute), Read data, Write data
- This will be compared to the access permissions stored in the MMU and catch any violation
  - The stack area can be set for **N**o-e**X**ecute (**NX or X=0**)
  - If the processor sees an attempt to execute code from the stack it will halt the program



http://ece-research.unm.edu/jimp/310/slides/micro_arch2.html

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
- Use of ASLR and No-eXecute controls can greatly reduce the risk of exploiting buffer overflow vulnerabilities
- Beyond code-injection techniques, alternative exploits such as return-oriented programming are explored in our projects

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

STACK FRAMES

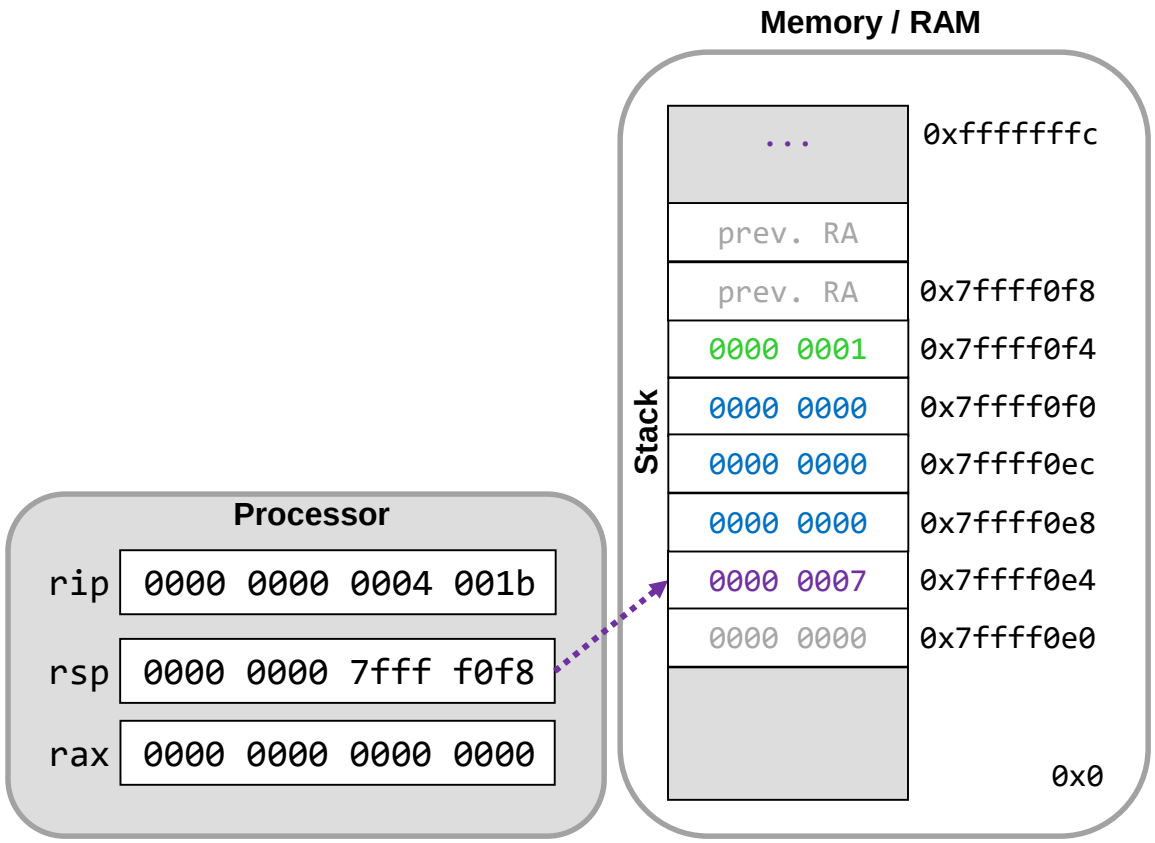
Stack Frame Motivation 1

CS:APP 3.10.5

- Under certain circumstances the compile cannot easily generate code using the stack pointer (%rsp) alone
  - The most common of these cases is when the needed 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?
```

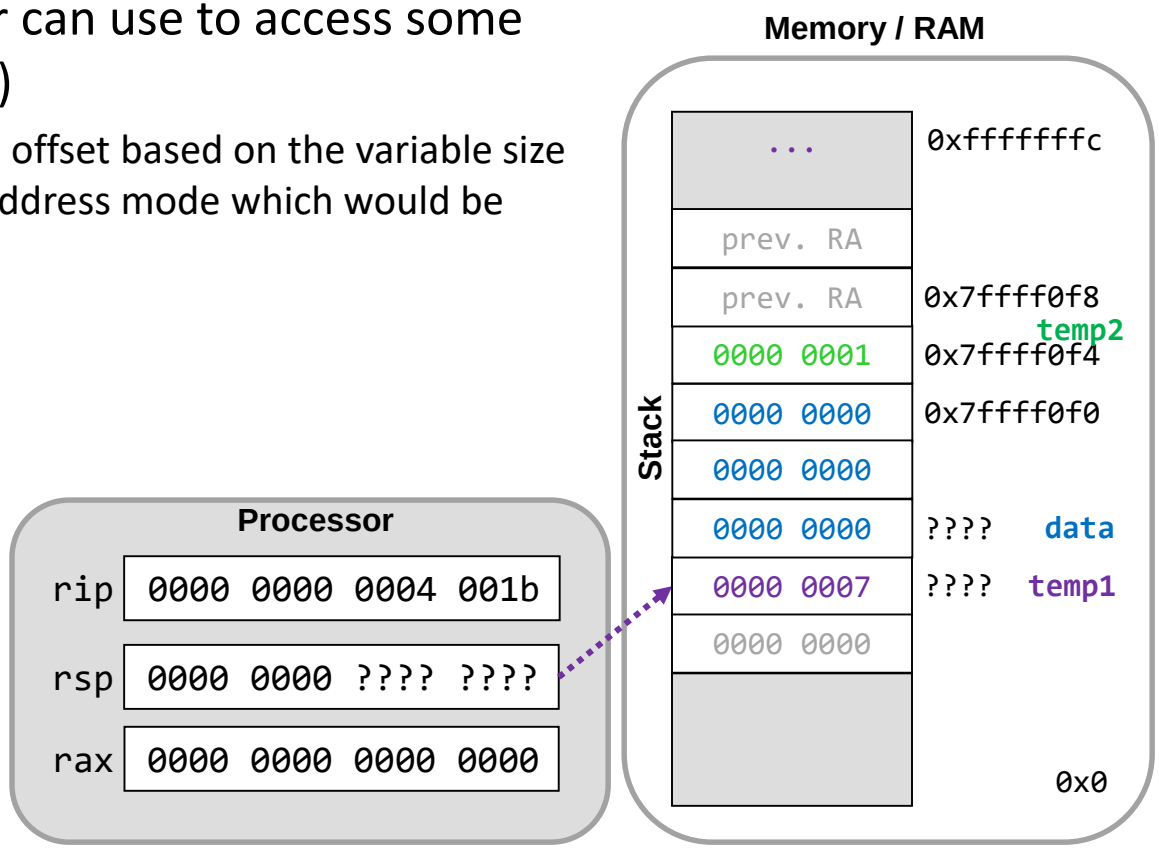


Stack Frame Motivation 2

- 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?
```



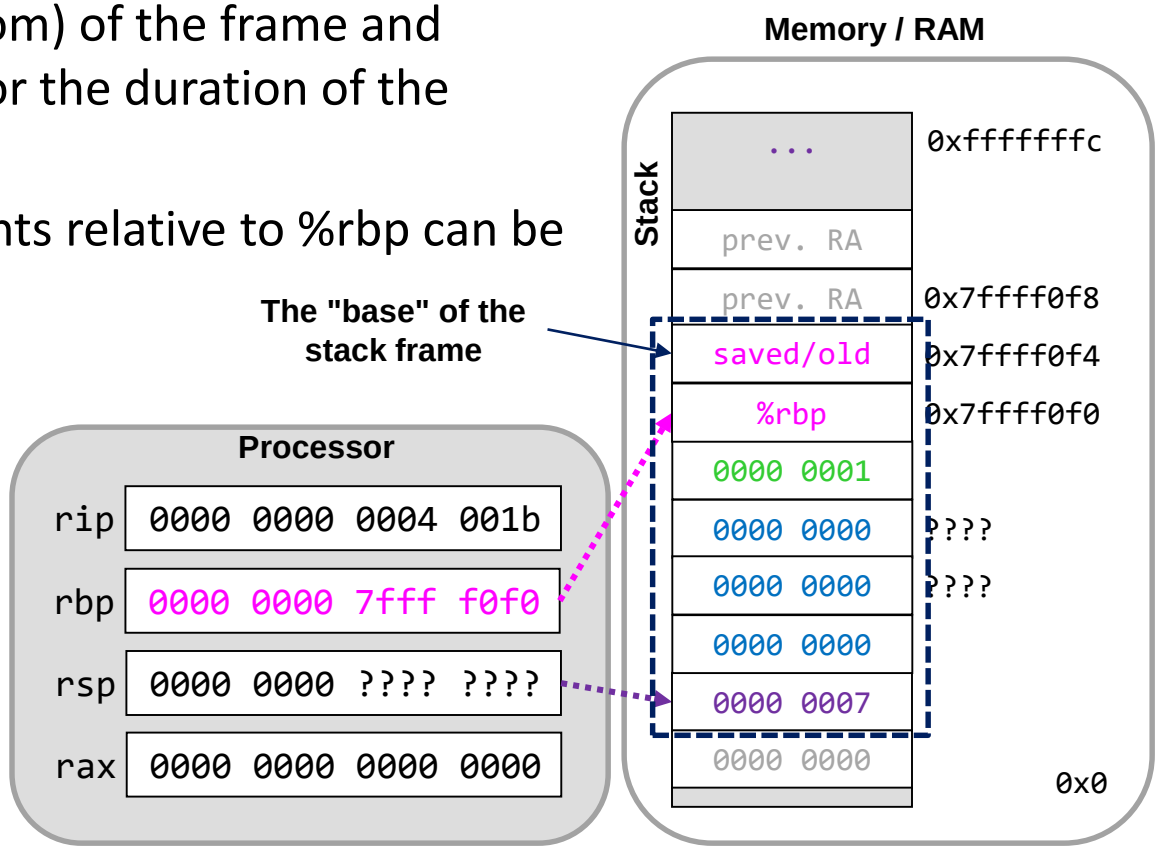
Base/Frame Pointer

- 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;
  ...
}
```

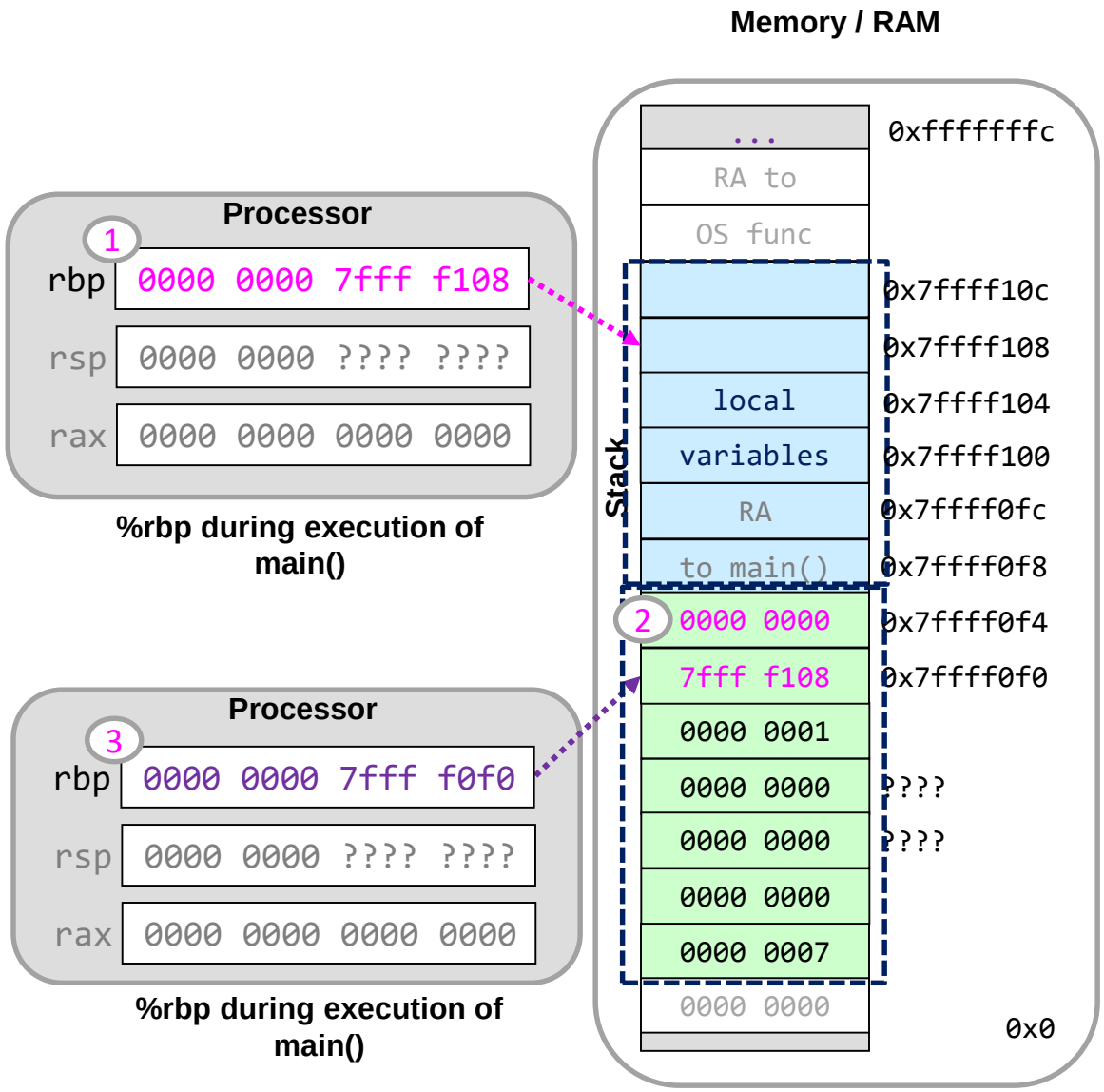
```
movl  (%rsp), %eax    # access temp1
movl  4(%rsp), %ecx   # access data[0]
movl  -4(%rbp), %edx  # access temp2
```



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
```

Processor

1

rbp	0000 0000 7fff f108
rsp	0000 0000 7fff f0f8
rax	0000 0000 0000 0000

Processor

2

rbp	0000 0000 7fff f0f0
rsp	0000 0000 ???? ????
rax	0000 0000 0000 0000

Memory / RAM