CS 88: Security and Privacy

07: Software Security: Attacks and Defenses

09-20-2022

SWARTHMORE COLLEGE

Format String Vulnerabilities

Variable arguments in C

In C, we can define a function with a variable number of arguments

```
void printf(const char* format,….)
```

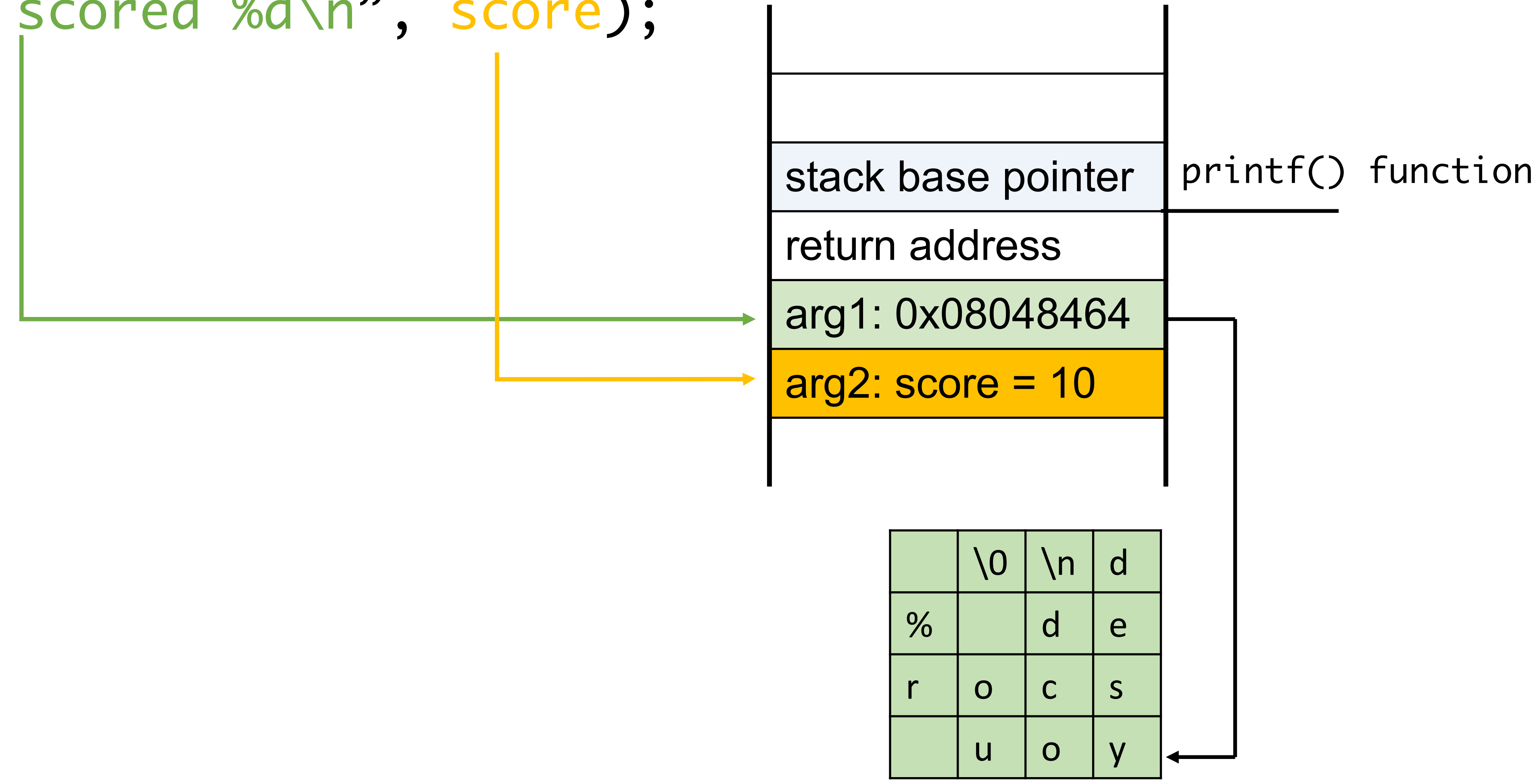
Usage:

```
printf(“hello world”);
printf(“length of %s = %d \n”, str, str.length());
```

format specification encoded by special % characters

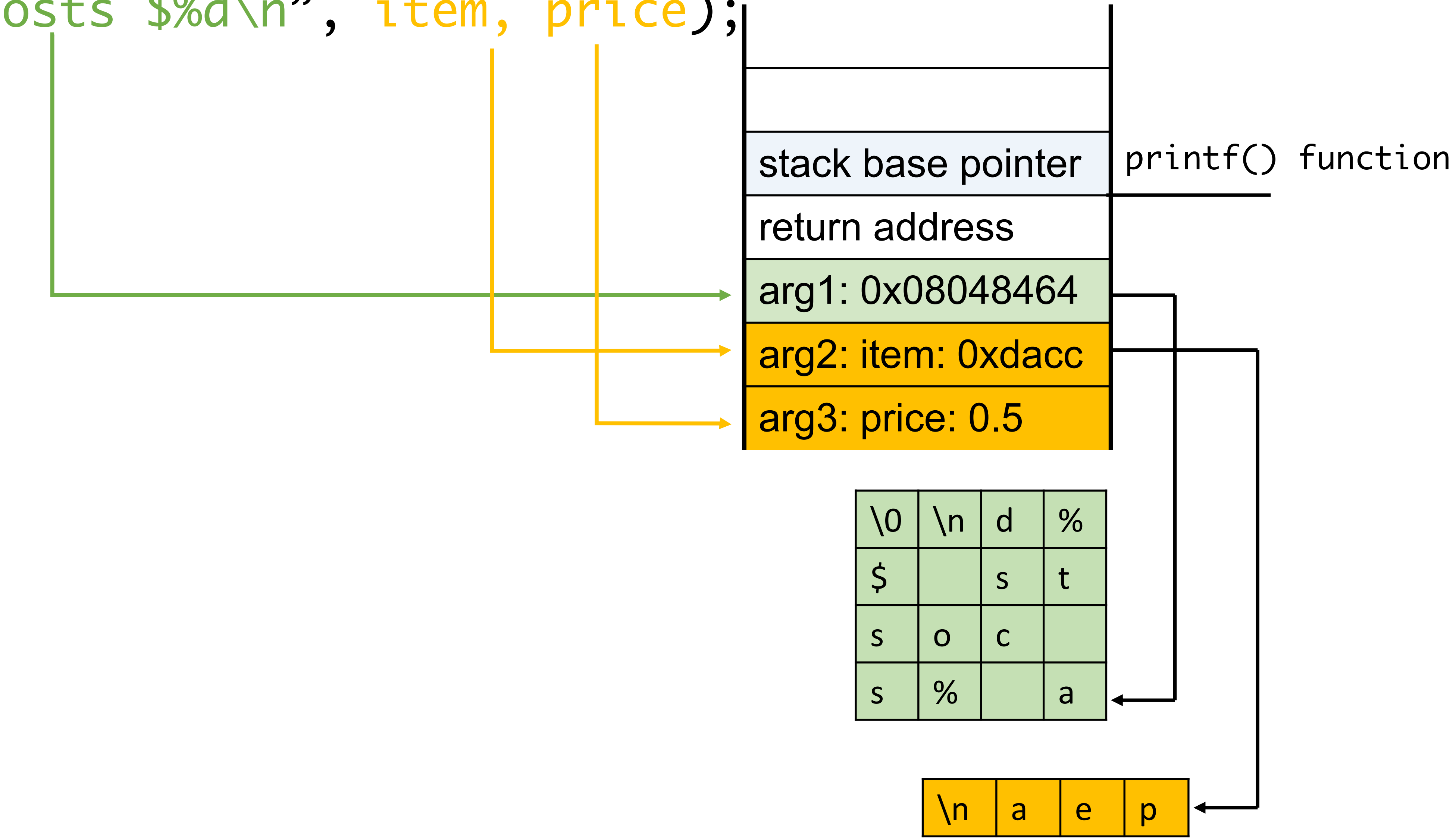
fun with format strings

```
printf("you scored %d\n", score);
```



fun with format strings

```
printf("a %s costs $%d\n", item, price);
```



Implementation of printf

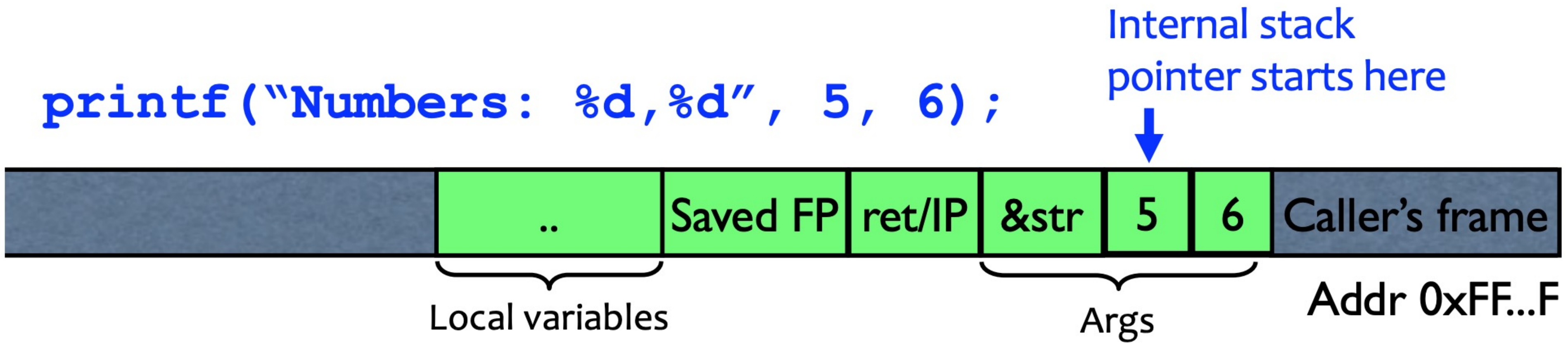
- Special functions va_start, va_arg, va_end
  compute arguments at run-time

```
void printf(const char* format, ...)
{
     int i; char c; char* s; double d;
     va_list ap; /* declare an "argument pointer" to a variable arg list */
     va_start(ap, format);  /* initialize arg pointer using last known arg */

     for (char* p = format; *p != '\0'; p++) {
        if (*p == '%') {
           switch (*++p) {
              case 'd':
                 i = va_arg(ap, int); break;
              case 's':
                 s = va_arg(ap, char*); break;
              case 'c':
                 c = va_arg(ap, char); break;
              }
              ... /* etc. for each % specification */
           }
     }
     ...

     va_end(ap);  /* restore any special stack manipulations */
}
```

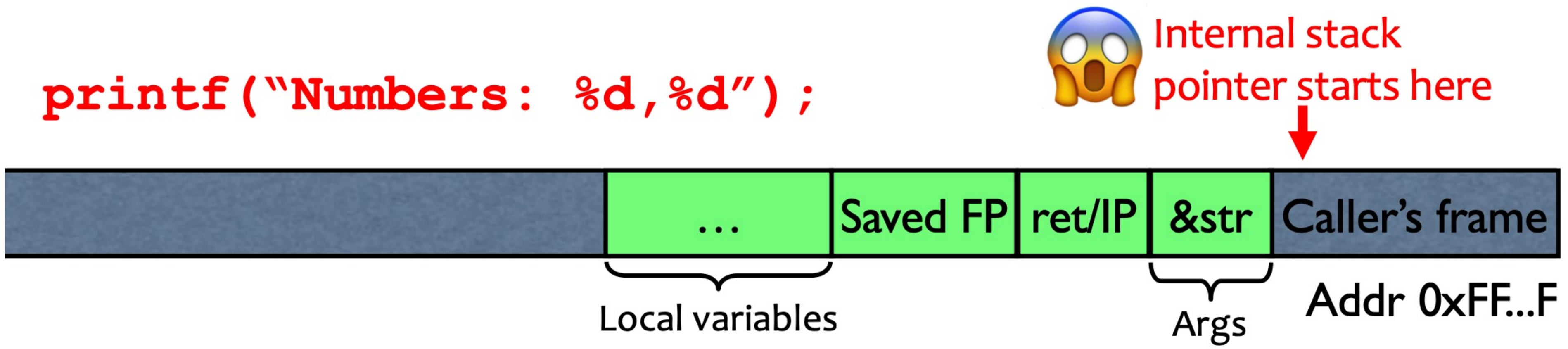
printf has an internal stack pointer

Closer look at the stack

```
printf("Numbers: %d,%d", 5, 6);
```



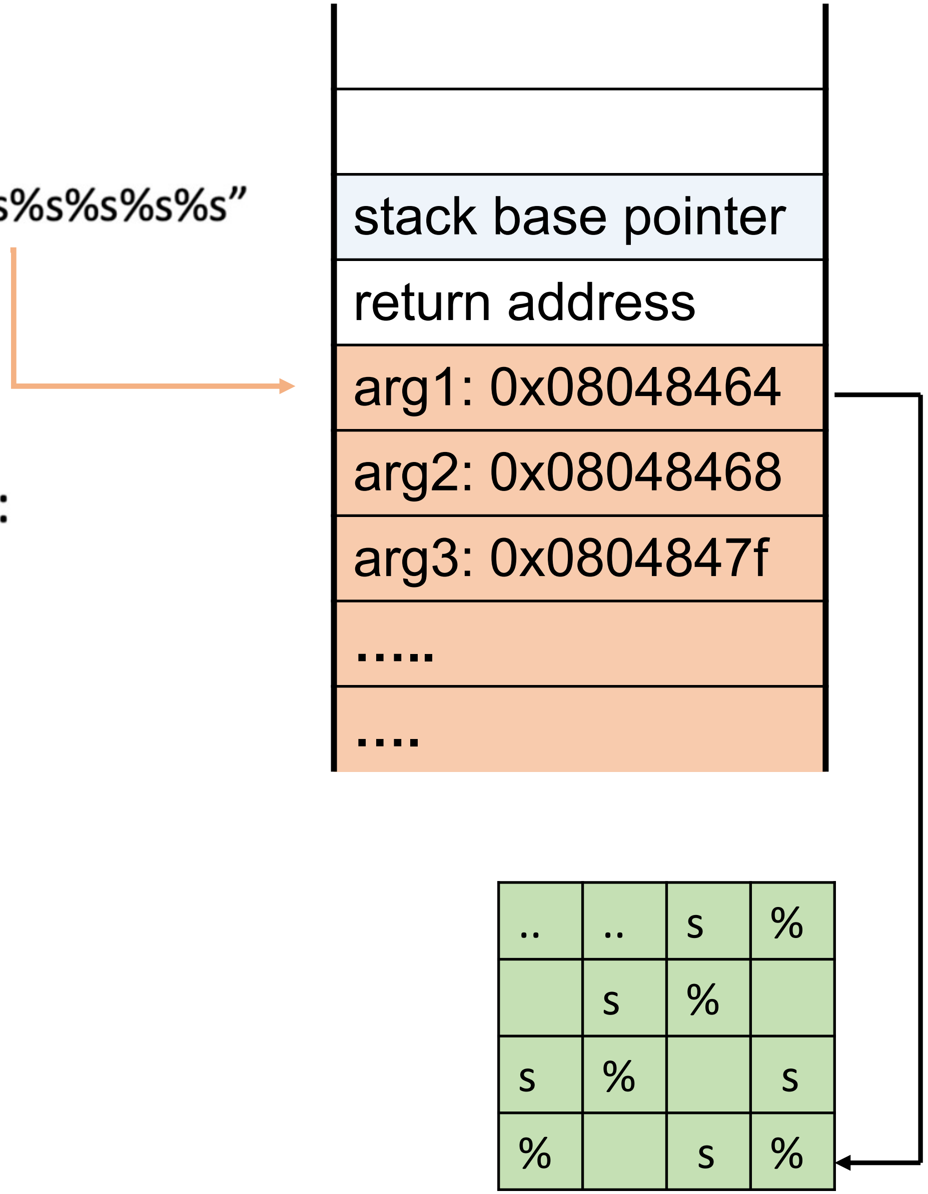

```
printf("Numbers: %d,%d");
```



Sloppy use of printf

```
void main(int argc, char* argv[])
{
    printf( argv[1] );
}
```

argv[1] = “%s%s%s%s%s%s%s%s%s%s%s”



Attacker controls format string gives all sorts of control:

- Print stack contents
- Print arbitrary memory
- Write to arbitrary memory

Format specification encoded by special % characters

Format Specifiers

Parameter	Meaning	Passed as
%d	decimal (int)	value
%u	unsigned decimal (unsigned int)	value
%x	hexadecimal (unsigned int)	value
%s	string ((const) (unsigned) char *)	reference
%n	number of bytes written so far, (* int)	reference

The %n format specifier

- `%n` format symbol tells `printf` to write the number of characters that have been printed
  - Argument of `printf` is interpreted as a destination address
- `printf (“overflow this!%n”, &myVar);`
  - Writes 14 into `myVar`.

The %n format specifier

- `%n` format symbol tells `printf` to write the number of characters that have been printed
  - Argument of `printf` is interpreted as a destination address
- `printf (“overflow this!%n”, &myVar);`
  - Writes 14 into `myVar`.
- What if printf does not have an argument?
  - `char buf[16] = “Overflow this!%n”;`
  - `printf(buf);`

A. Store the value 14 in buf
B. Store the value 14 on the stack (specify where)
C. Replace the string Overflow with 14
D. Something else

The %n format specifier

- `%n` format symbol tells `printf` to write the number of characters that have been printed
  - Argument of `printf` is interpreted as a destination address

- `printf (“overflow this!%n”, &myVar);`
  - Writes 14 into `myVar`.

- What if printf does not have an argument?

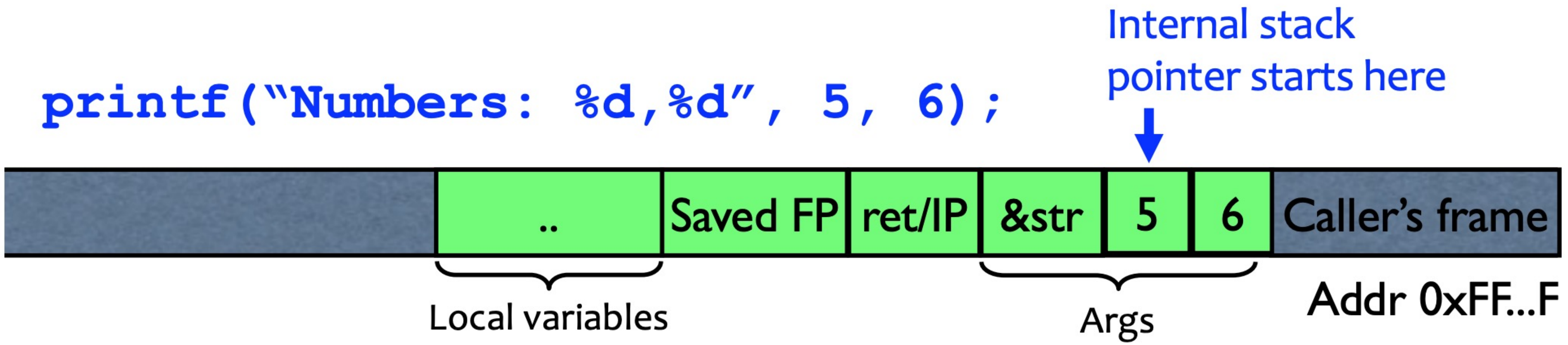
  - `char buf[16] = “Overflow this!%n”;`
  - `printf(buf);`

- Stack location pointed to by printf’s internal stack pointer will be interpreted as an address

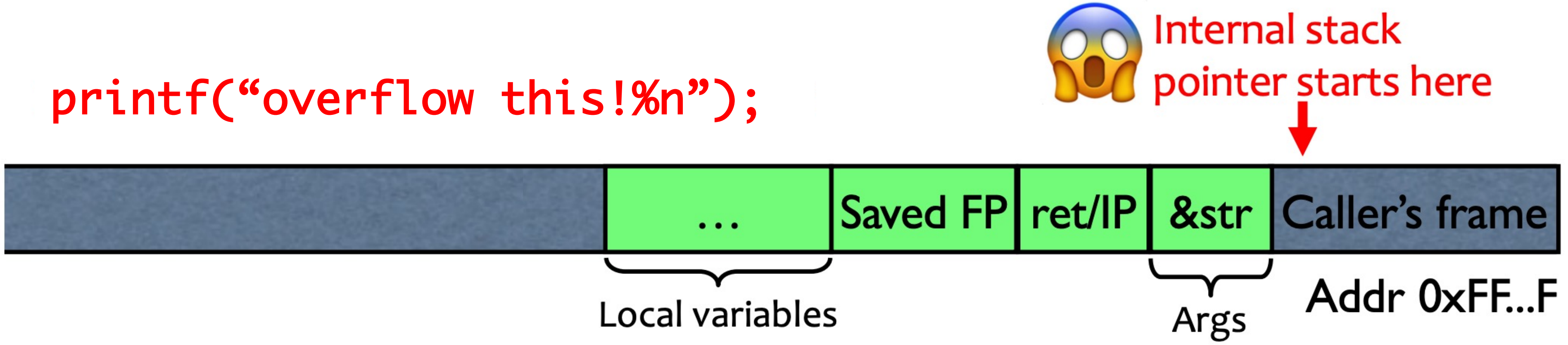
- Write # characters at this address

Closer look at the stack

```
printf("Numbers: %d,%d", 5, 6);
```




```
printf("overflow this!%n");
```



Write 14 into the caller's frame!

fun with printf: what's the output of the following statements?

```
printf("100% dive into C!")

printf("100% samy worm");

printf("%d %d %d %d");

printf("%d %s);

printf("100% not another segfault!");
```

fun with printf: what's the output of the following statements?

```
printf("100%dive into C!")
```

100 + value 4 bytes below retaddress as an integer + "ive"

```
printf("100%samy worm");
```

prints bytes pointed to by the stack entry up through the first NULL

```
printf("%d %d %d %d");
```

print series of stack entries as integers

```
printf("%d %s);
```

print value 4 bytes below return address plus bytes pointed to by the preceding stack entry

```
printf("100% not another segfault!");
```

prints 100 not another segfault! and stores the number 3 on the stack

Viewing the stack

We can show some parts of the stack memory by using a format string like this:

C code

```
printf ("%08x.%08x.%08x.%08x.%08x\n");
```

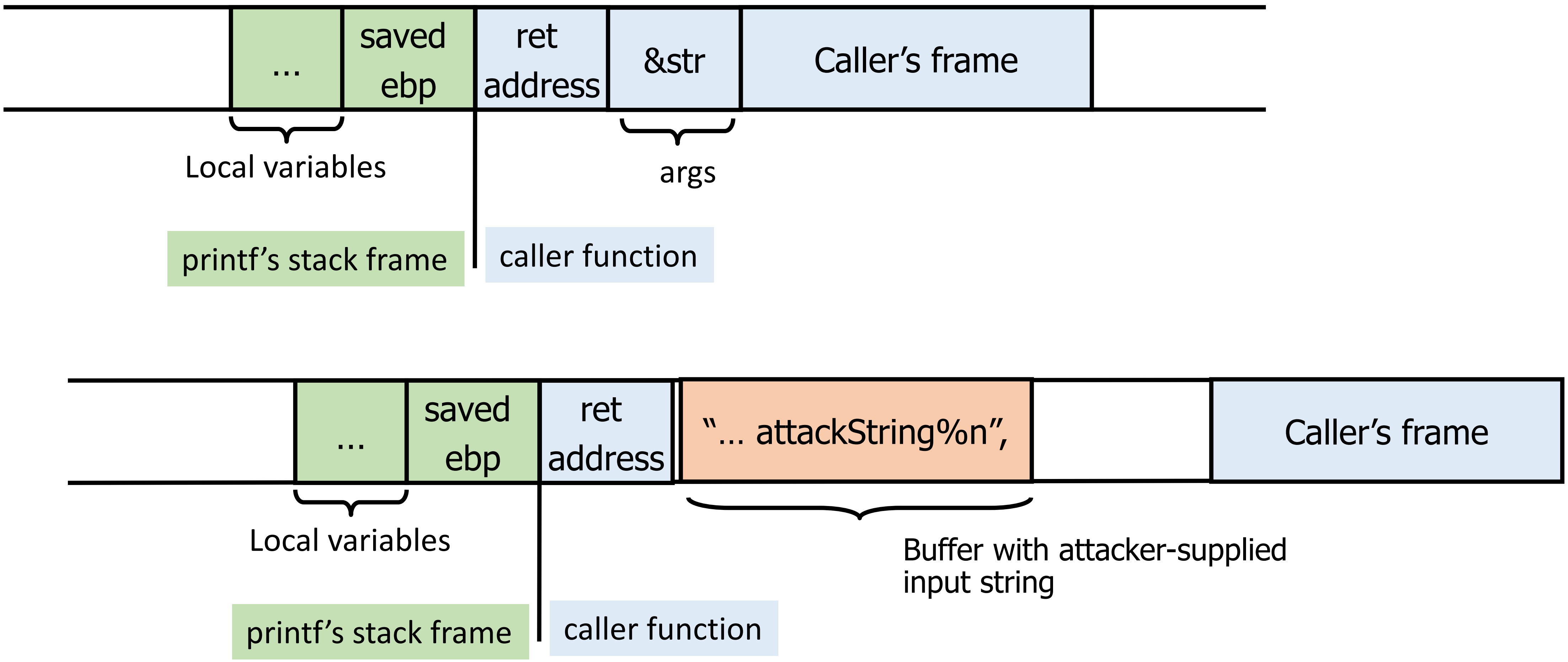
Output

```
40012980.080628c4.bffff7a4.00000005.08059c04
```

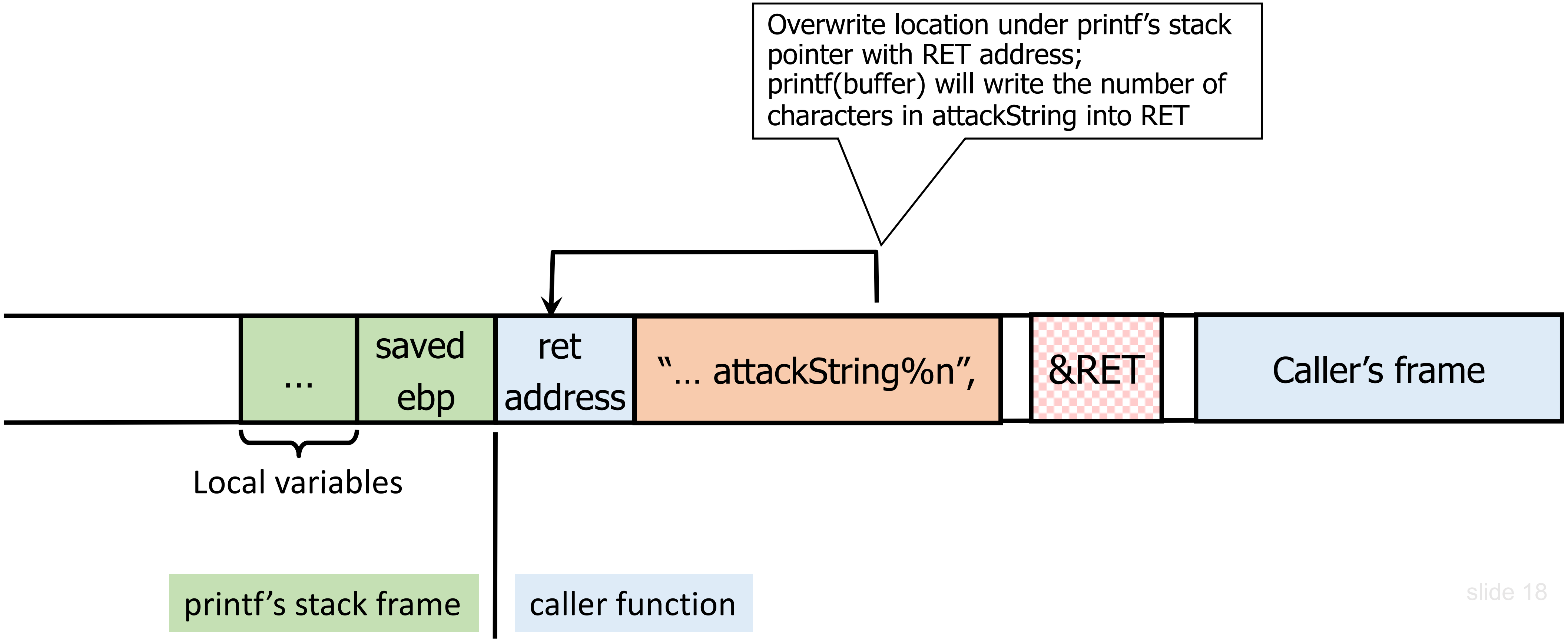
instruct printf:

- retrieve 5 parameters
- display them as 8-digit padded hexademical numbers

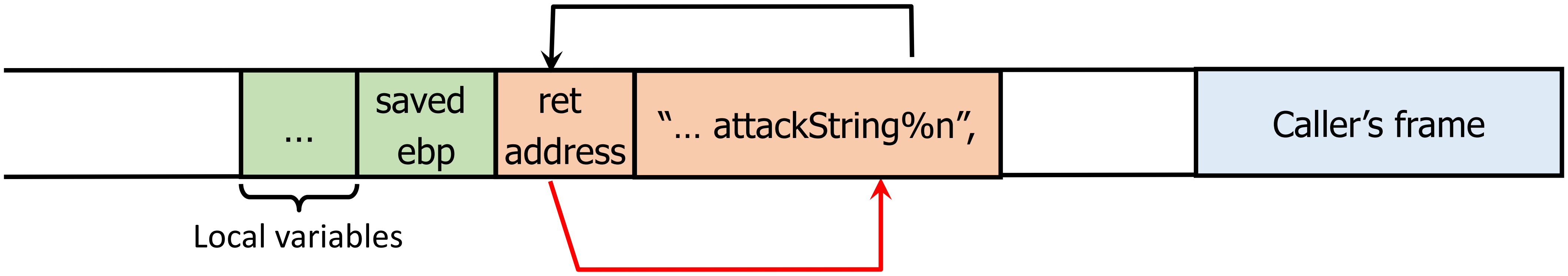
Using %n to Mung Return Address



Using %n to Mung Return Address



Using %n to Mung Return Address



C has a concise way of printing multiple symbols:

- %Mx will print exactly 4M bytes (taking them from the stack).
- Attack string should contain enough "%Mx" so that the number of characters printed is equal to the most significant byte of the address of the attack code.
- Repeat three times (four "%n" in total) to write into &RET+1, &RET+2, &RET+3, thus replacing RET with the address of attack code byte by byte.

See "Exploiting Format String Vulnerabilities" for details

If your program has a format string bug, assume that the attacker can learn all secrets stored in memory, and assume that the attacker can take control of your program.

Validating input

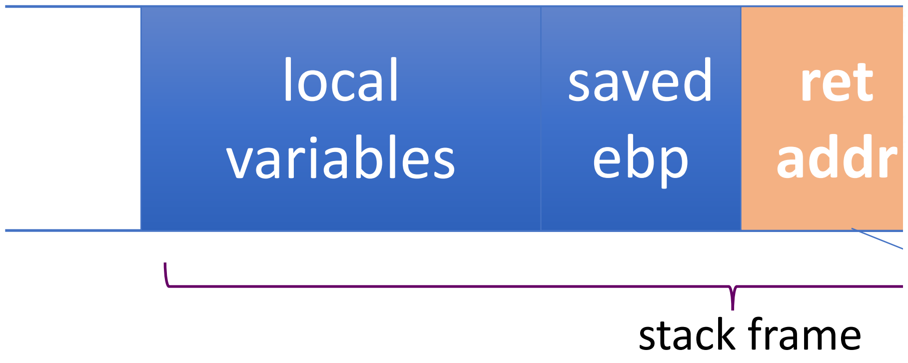
- Determine acceptable input, check for match --- don’t just check against list of “non-matches”
- Limit maximum length
- Watch out for special characters, escape chars.
- Check bounds on integer values
- Check for negative inputs
- Check for large inputs that might cause overflow!

Validating input

- Filenames
- Command-line arguments
- Even argv[0]…
- Commands
  - E.g., URLs, http variables., SQL
  - E.g., cross site scripting, (next lecture)

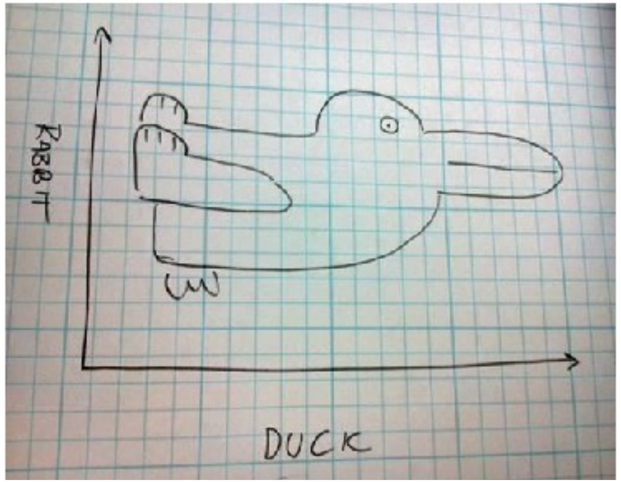
Memory attacks

The problem: mixing data with control flow in memory



Your program manipulates data

Data manipulates your program



Memory Attacks: Causes

"Classic" memory exploit involves code injection

- malicious code @ predictable location in memory -> masquerading as data
- trick vulnerable program into passing control

Memory Attacks: Causes and Cures

"Classic" memory exploit involves code injection

Idea: prevent execution of untrusted code

Developer approaches:

- Use of safer functions like strlcpy(), strlcat() etc.
- safer dynamic link libraries that check the length of the data before copying.

Hardware approaches: Non-Executable Stack

OS approaches: ASLR (Address Space Layout Randomization)

Compiler approaches: Stack-Guard Pro-Police

Data Execution Prevention: a.k.a Mark memory as non-executable

Each page of memory has separate access permissions:

- R -> Can Read, W -> Can Write, X -> Can Execute

Mark all writeable memory locations as non-executable

NX-bit on AMD64, **XD-bit** on Intel x86 (2005), **XN-bit** on ARM

- Now you can't write code to the stack or heap
- No noticeable performance impact

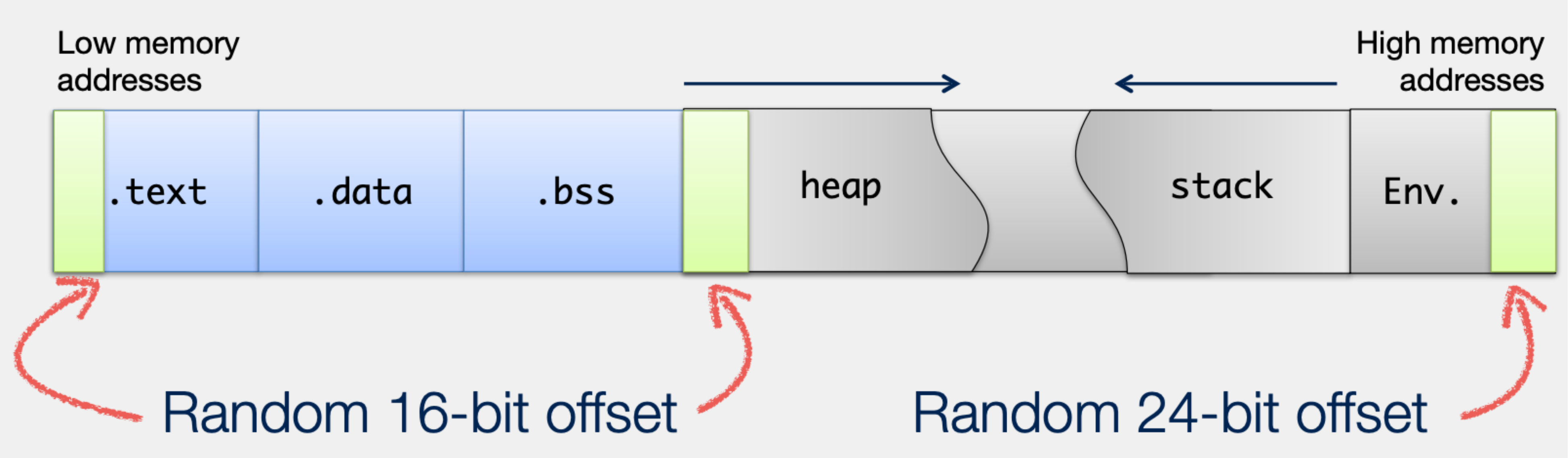
Address Space Layout Randomization

Onload: Randomly relocate the base address of everything in memory

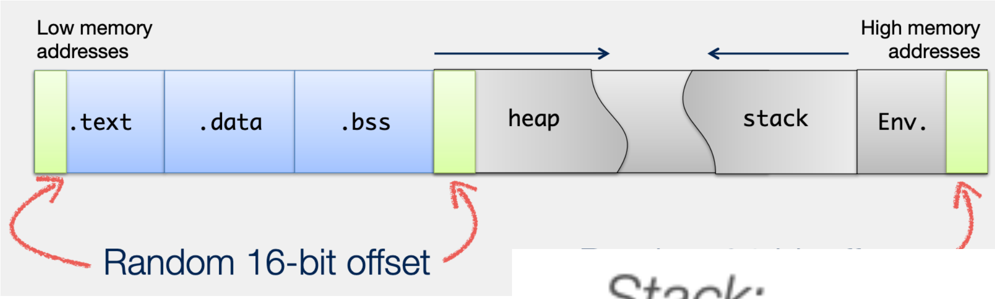
- libraries (DLLs, shared libs), application code, stack heap

⇒ attacker does not no location

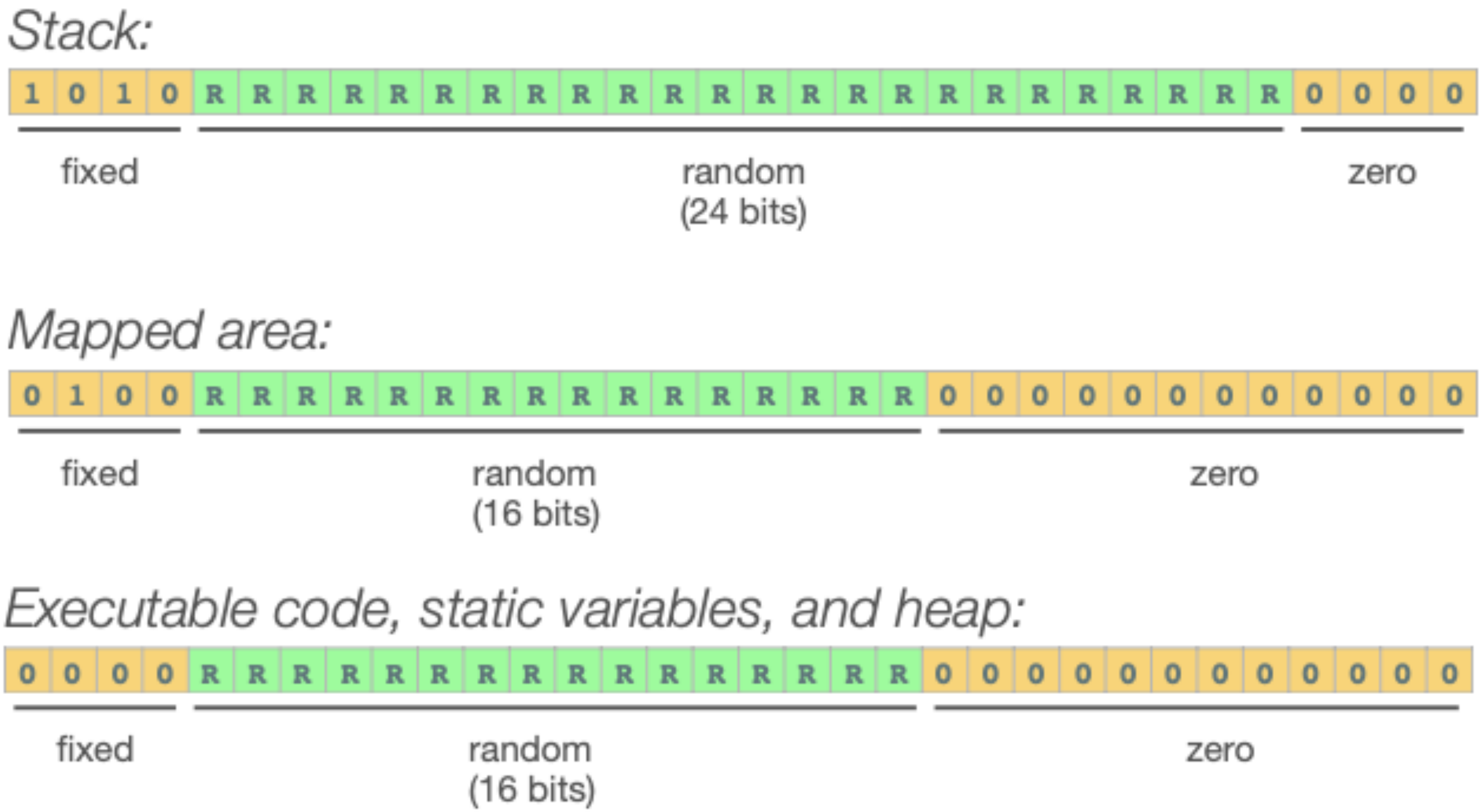
Example: PAX implementation

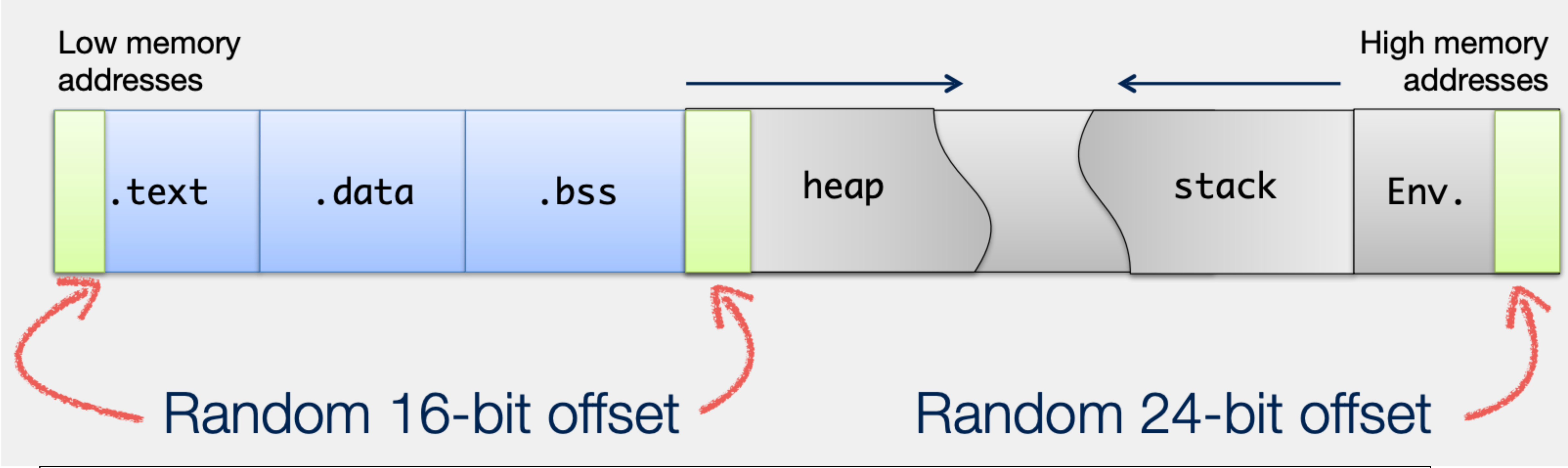


Address Space Layout Randomization



32-bit PaX ASLR (x86)





randomize the start location of stack, code data.

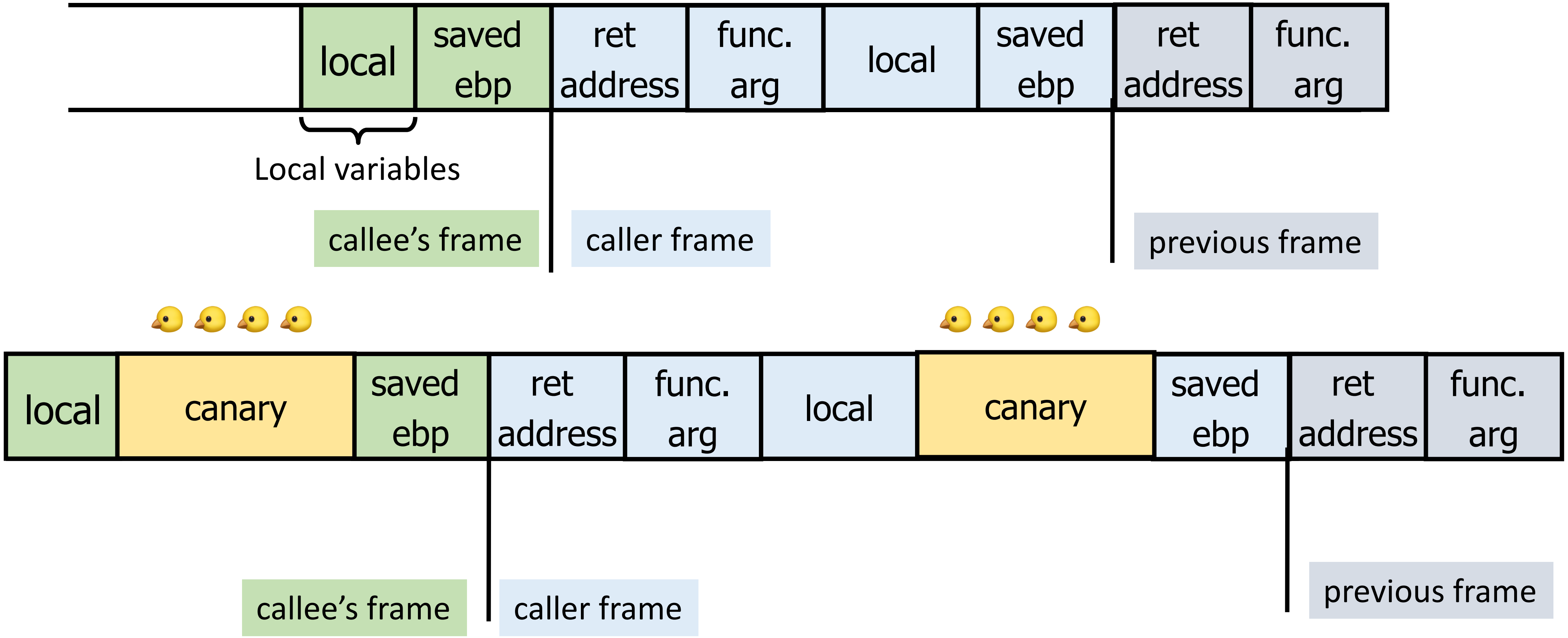
Launch buffer overflow? Difficult to guess the stack address!

Difficult to guess %ebp address and address of the malicious code

Compiler Defenses: Stack Canary

Method 1: StackGuard

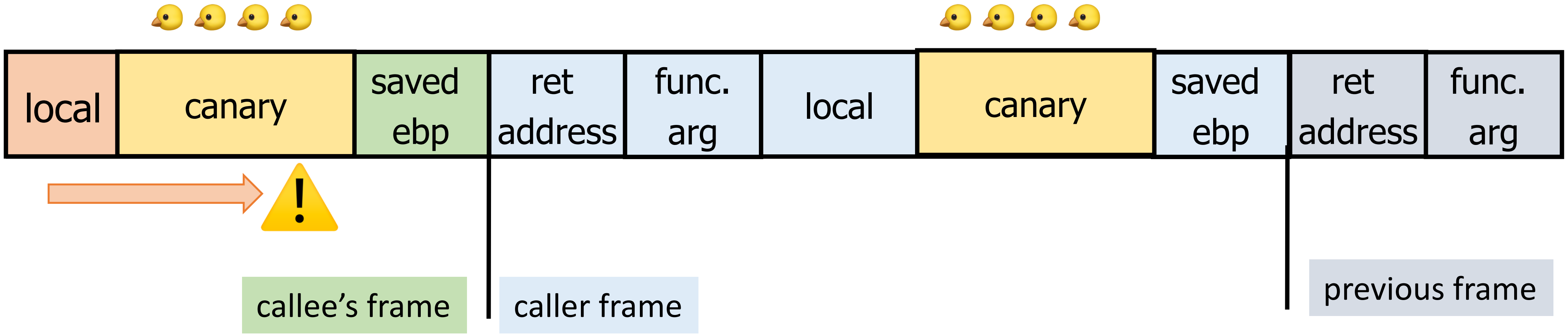
- Embed "canaries" (stack cookies) in stack frames and verify their integrity prior to function return.



StackGuard

Minimal performance effects: 8% for Apache
Program must be recompiled

Overflow canary? Segfault!



Random canary:

- Random string **chosen at program startup**
- To corrupt, attacker must learn/guess current random string

Terminator canary:

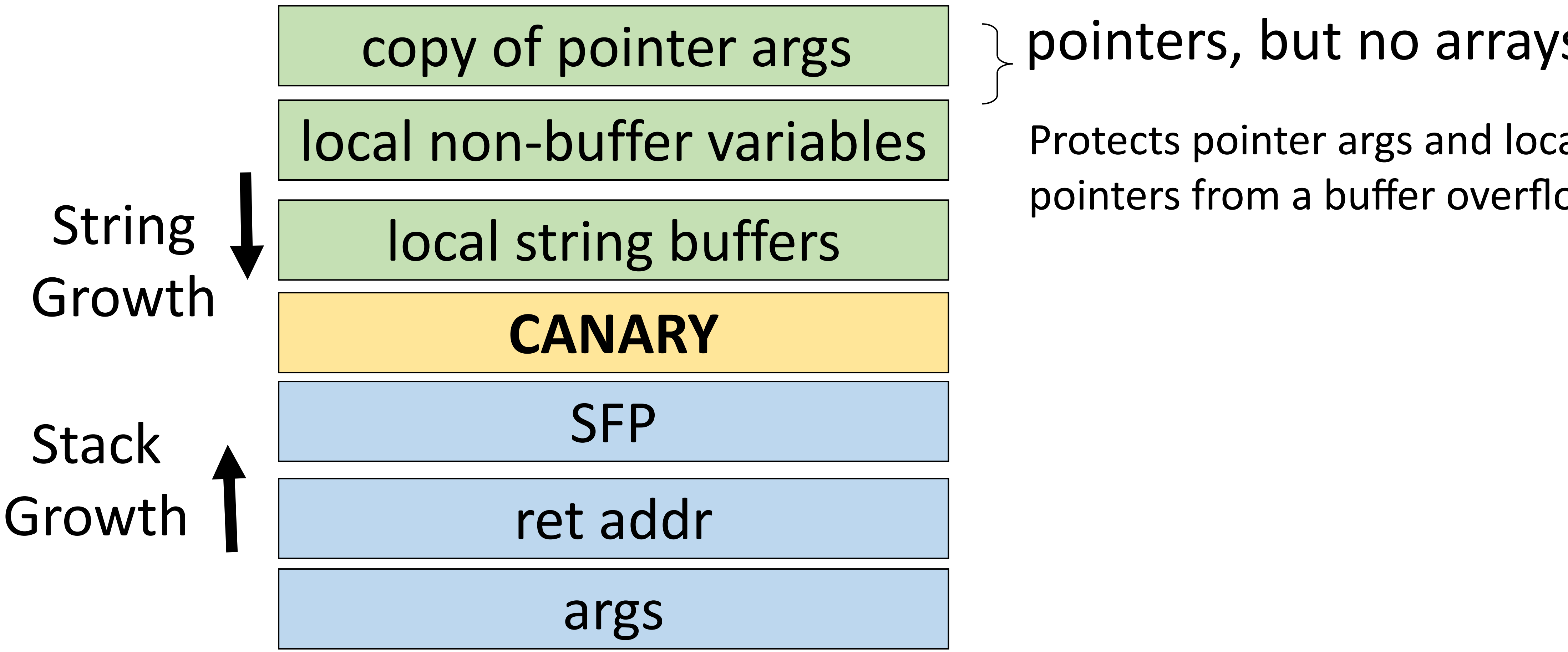
- {0, newline, linefeed, EOF}
- String functions will not copy beyond terminator
- Attacker cannot use string functions to corrupt the stack

Canary check in gcc:

```
Dump of assembler code for function foo:
   0x0000120d <+0>:     endbr32
   0x00001211 <+4>:     push   %ebp
   0x00001212 <+5>:     mov    %esp,%ebp
   0x00001214 <+7>:     push   %ebx
   0x00001215 <+8>:     sub    $0x24,%esp
   0x00001218 <+11>:    call   0x12b4 <__x86.get_pc_thunk.ax>
   0x0000121d <+16>:    add    $0x2db3,%eax
   0x00001222 <+21>:    mov    0x8(%ebp),%edx
   0x00001225 <+24>:    mov    %edx,-0x1c(%ebp)
   0x00001228 <+27>:    mov    %gs:0x14,%ecx
   0x0000122f <+34>:    mov    %ecx,-0xc(%ebp)
   0x00001232 <+37>:    xor    %ecx,%ecx
   0x00001234 <+39>:    sub    $0x8,%esp
   0x00001237 <+42>:    pushl  -0x1c(%ebp)
   0x0000123a <+45>:    lea    -0x18(%ebp),%edx
   0x0000123d <+48>:    push   %edx
   0x0000123e <+49>:    mov    %eax,%ebx
   0x00001240 <+51>:    call   0x10a0 <strcpy@plt>
   0x00001245 <+56>:    add    $0x10,%esp
   0x00001248 <+59>:    nop
   0x00001249 <+60>:    mov    -0xc(%ebp),%eax
   0x0000124c <+63>:    xor    %gs:0x14,%eax
   0x00001253 <+70>:    je     0x125a <foo+77>
   0x00001255 <+72>:    call   0x1340 <__stack_chk_fail_local>
   0x0000125a <+77>:    mov    -0x4(%ebp),%ebx
   0x0000125d <+80>:    leave
   0x0000125e <+81>:    ret
End of assembler dump.
```

StackGuard Variations

- Rearrange stack layout to prevent ptr overflow.

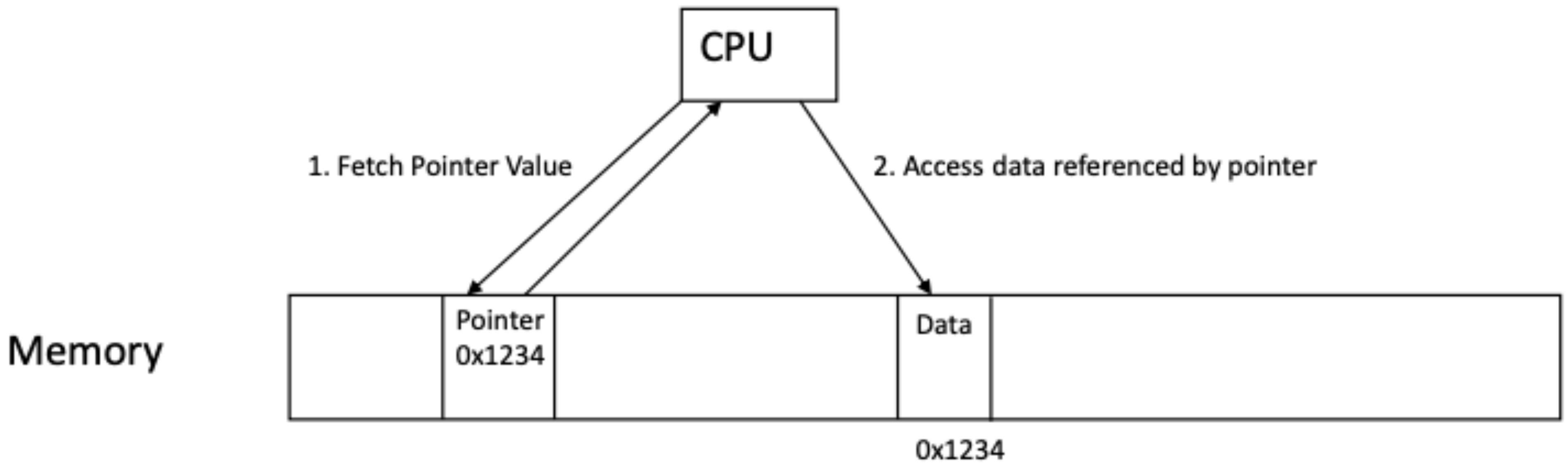


Protects pointer args and loca
pointers from a buffer overflo

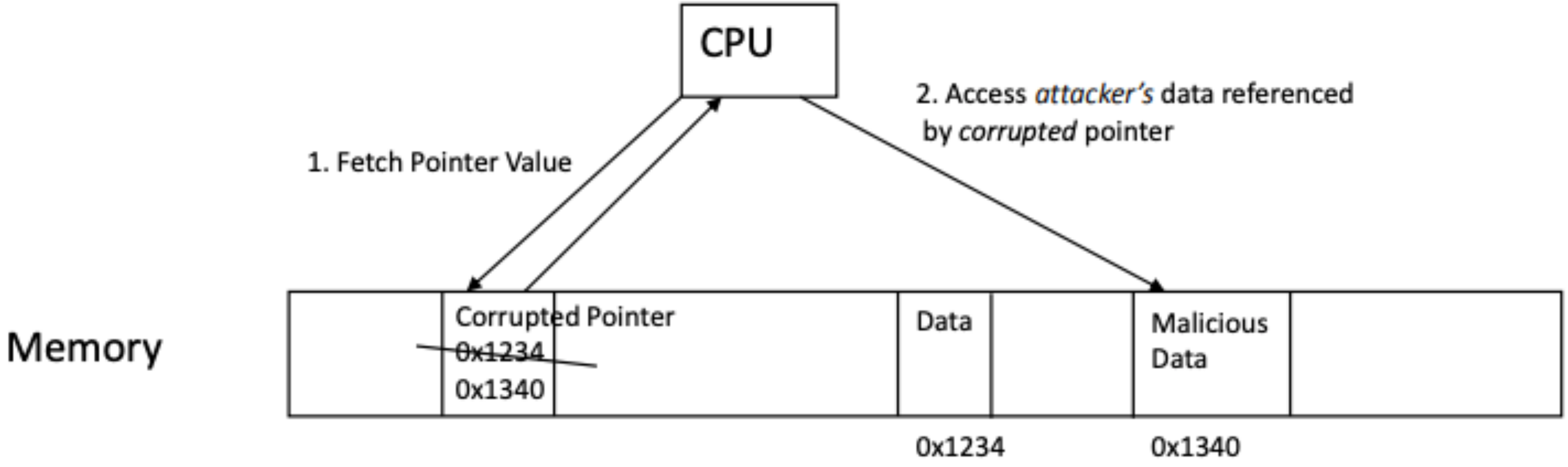
PointGaurd

- Insight:
  - pointers in memory corrupted via overflow
  - pointers in registers are not overflowable
- Solution:
  - Store pointers encrypted in memory
  - To dereference a pointer: decrypt it as you load it unto a register

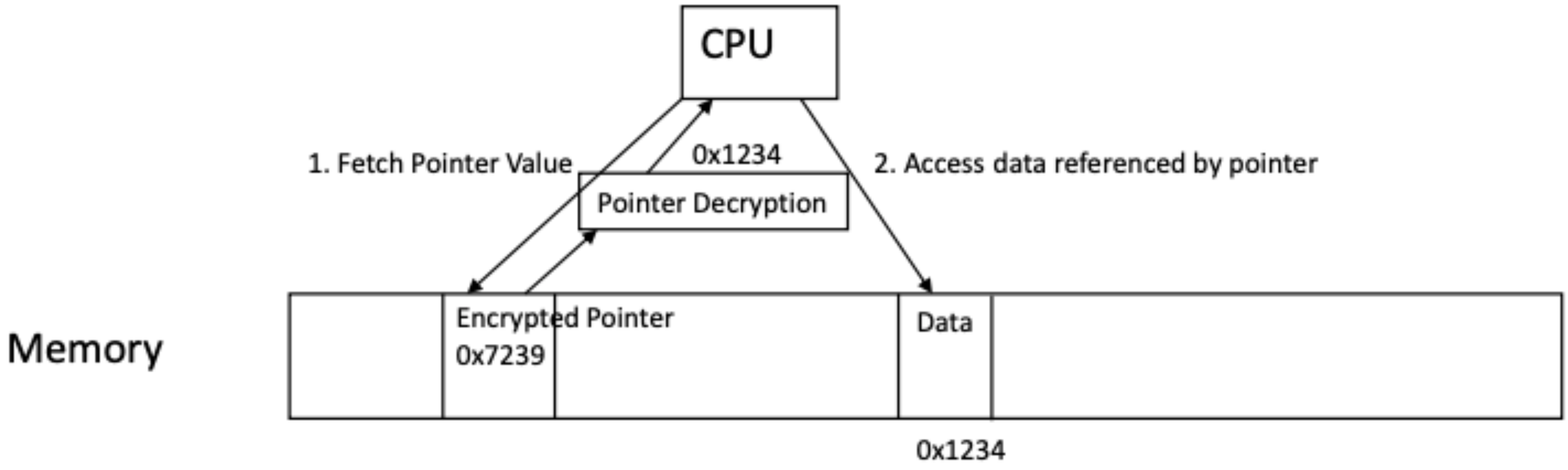
Normal Pointer Dereference



Normal Pointer Dereference under attack



PointerGuard Pointer Dereference



PointerGuard Pointer Dereference Under Attack