Lecture 08 – Format string vulnerabilities

Stephen Checkoway
Oberlin College

Goal

- Take control of the program (as usual)
- How?
  - Write4 (write 4 bytes to an arbitrary location)
  - Inject shellcode (or other exploits) into the process

What should we overwrite?

- Saved instruction pointer/return address (rip) on the stack
- Other pointers to code (we'll come back to this)

printf operation

- printf takes a format string and arguments
- printf copies the format string to its output, replacing conversion specifiers with values determined by the arguments
- Arguments are (normally) accessed one at a time, in turn
- Internally, printf keeps a pointer to the next argument to be converted by a conversion specifier
- Example: `printf("value = %d %c”, 42, 'm');`
  prints: `value = 42 m`

Common conversion specifiers

`%c`	Character	`%s`	String
`%d, %i`	Integer	`%p`	Pointer
`%u`	Unsigned integer	`%%`	Literal %
`%x, %X`	Hex	`%n`	Stores number of characters written
`%e, %f,`	Double		

printf family

- printf
- fprintf
- sprintf
- snprintf
- asprintf
- dprintf

- vprintf
- vfprintf
- vsprintf
- vsnprintf
- vasprintf
- vdprintf

These slides depict 32-bit x86

Key differences between x86-64 and x86 (for our purposes)

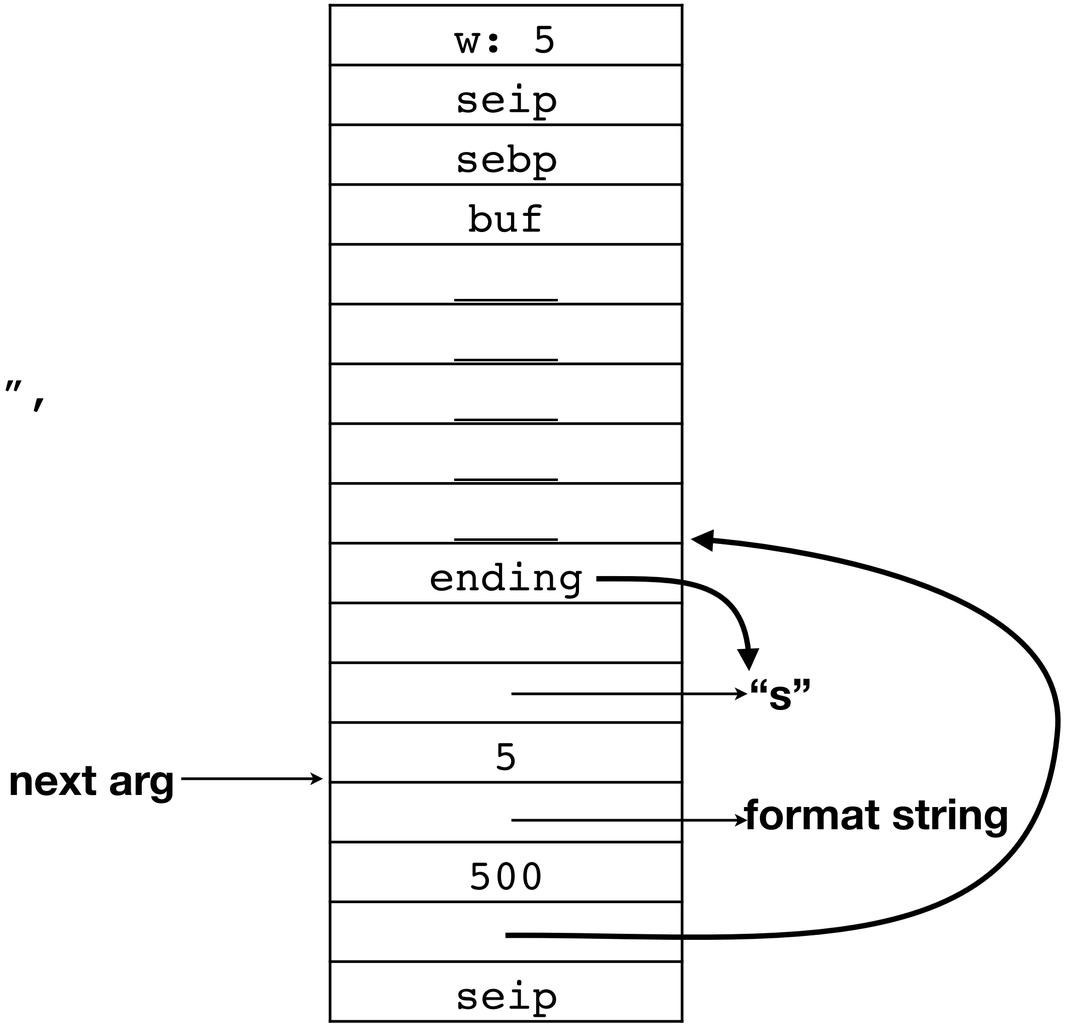
- Registers are 32 bits: eax, ebx, ecx, edx, edi, esi, ebp, esp
- Instruction pointer is 32-bits: eip
- Integers, longs, and pointers are 32 bits (ILP32 data model)
- Calling convention is that all arguments to functions are passed on the stack rather than the first six being passed in registers

The following slides use seip and sebp to mean the saved instruction pointer (eip) and frame pointer (ebp) on the stack

x86-64 is slightly more difficult to exploit due to addresses containing zeros; we'll return to this

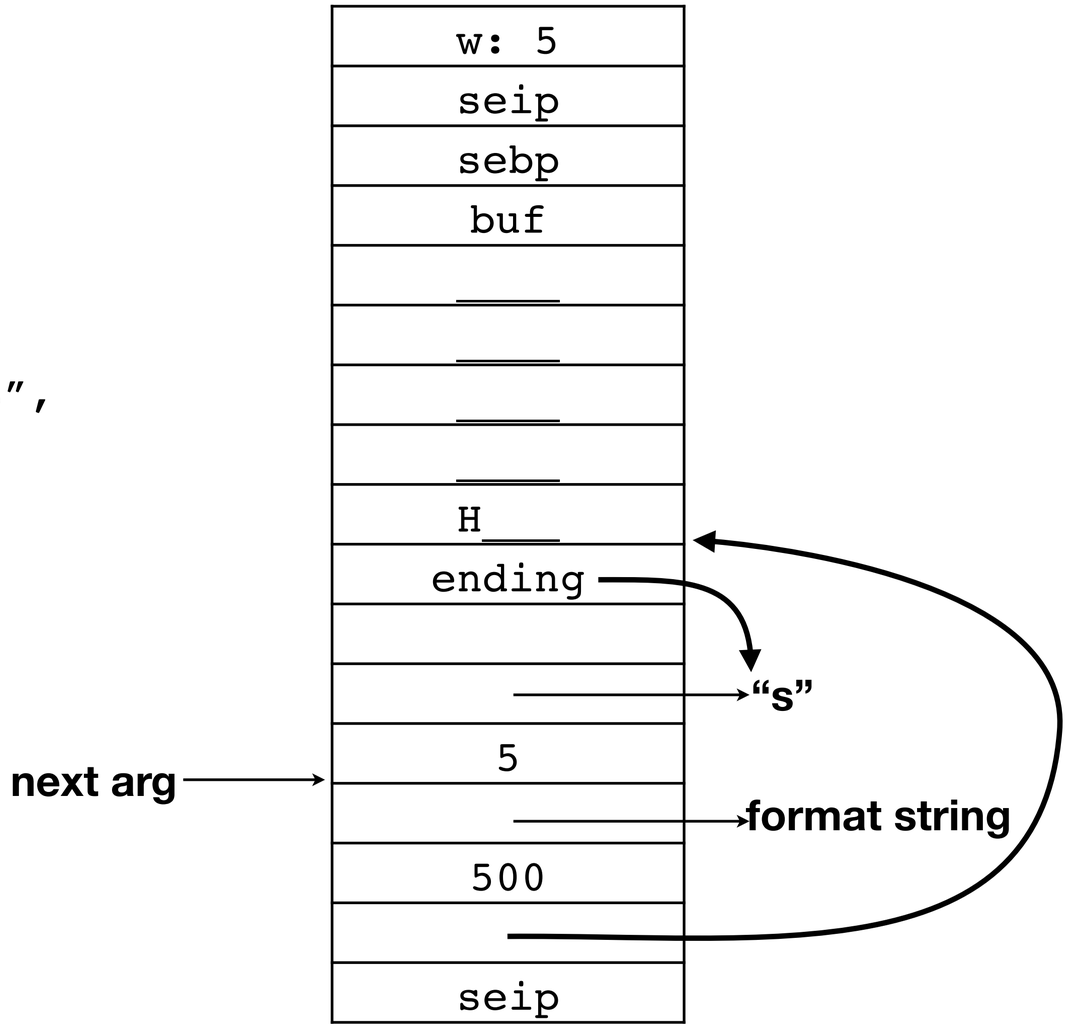
The way snprintf() normally works

```
void foo(int w) {
  char buf[500];
  const char *ending = w==1? “”:”s”;
  snprintf(buf, 500, “Hello %d world%s”,
         w, ending);
}
…
foo(5);
```



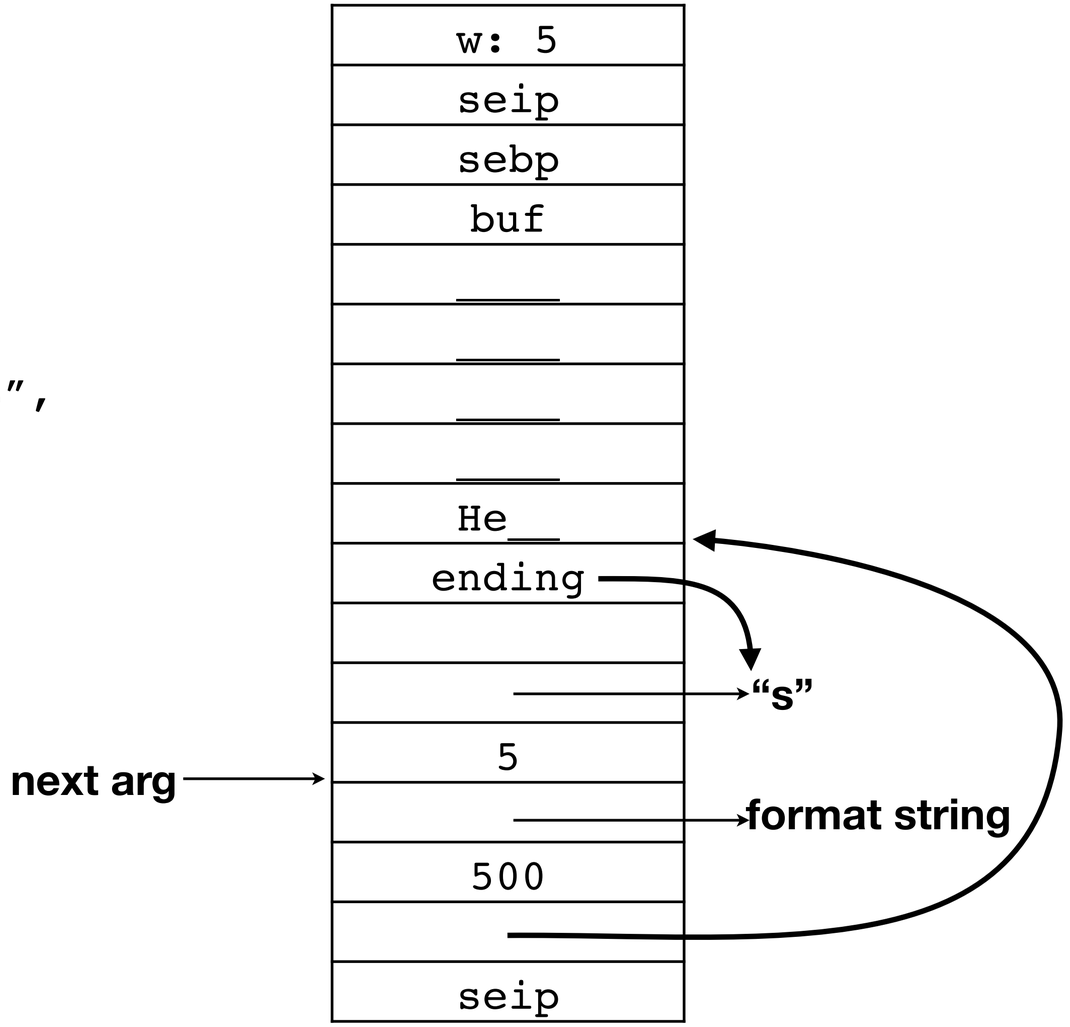
The way snprintf() normally works

```
void foo(int w) {
  char buf[500];
  const char *ending = w==1? “”:”s”;
  snprintf(buf, 500, “Hello %d world%s”,
         w, ending);
}
…
foo(5);
```



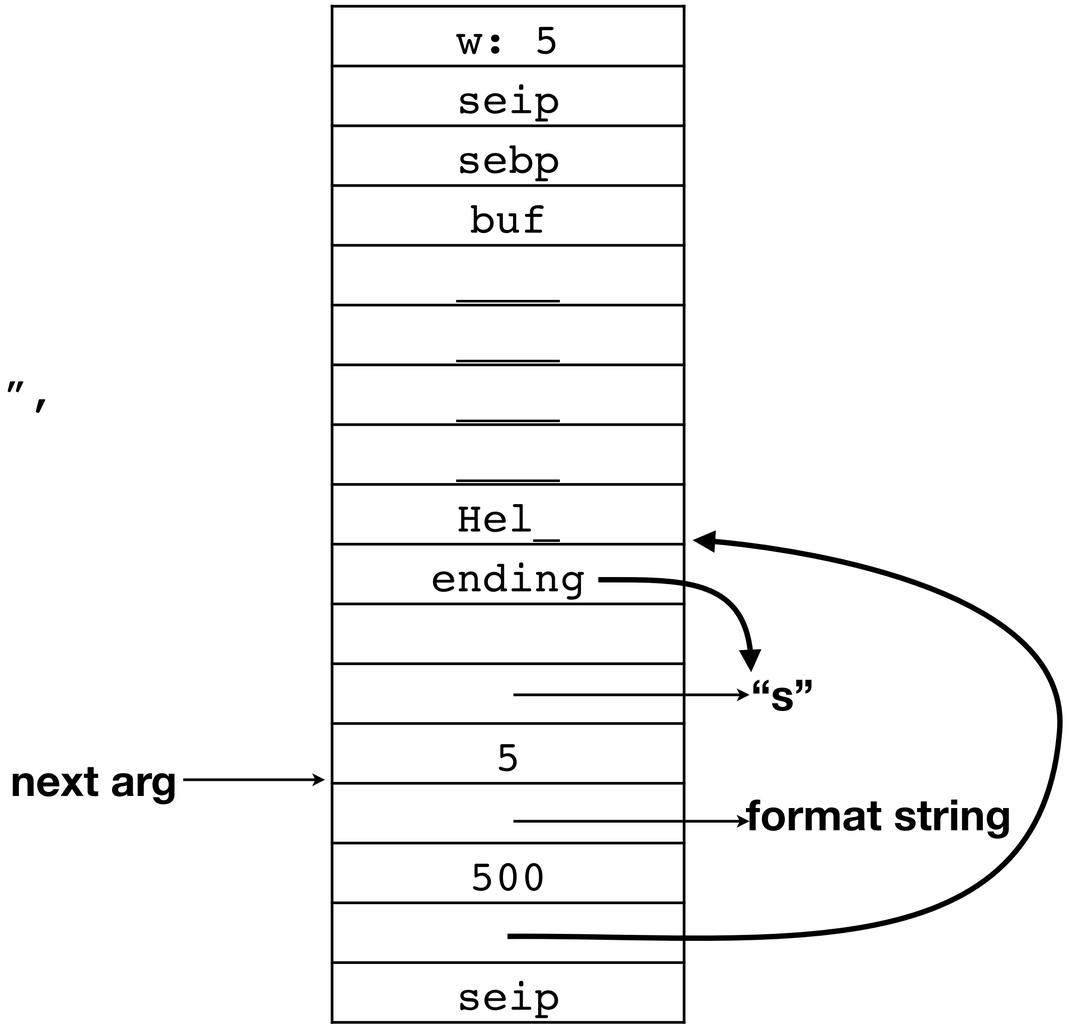
The way snprintf() normally works

```
void foo(int w) {
  char buf[500];
  const char *ending = w==1? “”:”s”;
  snprintf(buf, 500, “Hello %d world%s”,
         w, ending);
}
…
foo(5);
```



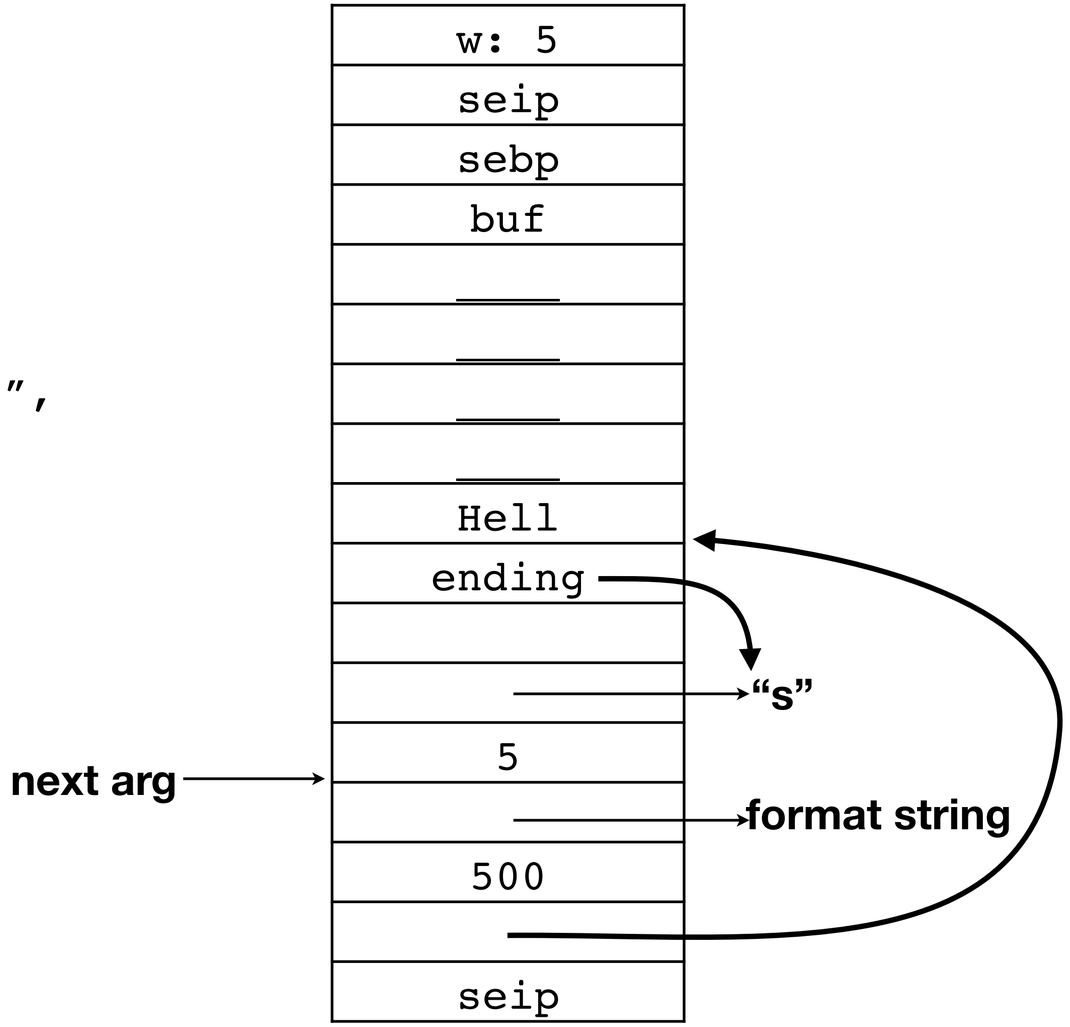
The way snprintf() normally works

```
void foo(int w) {
  char buf[500];
  const char *ending = w==1? “”:”s”;
  snprintf(buf, 500, “Hello %d world%s”,
         w, ending);
}
…
foo(5);
```



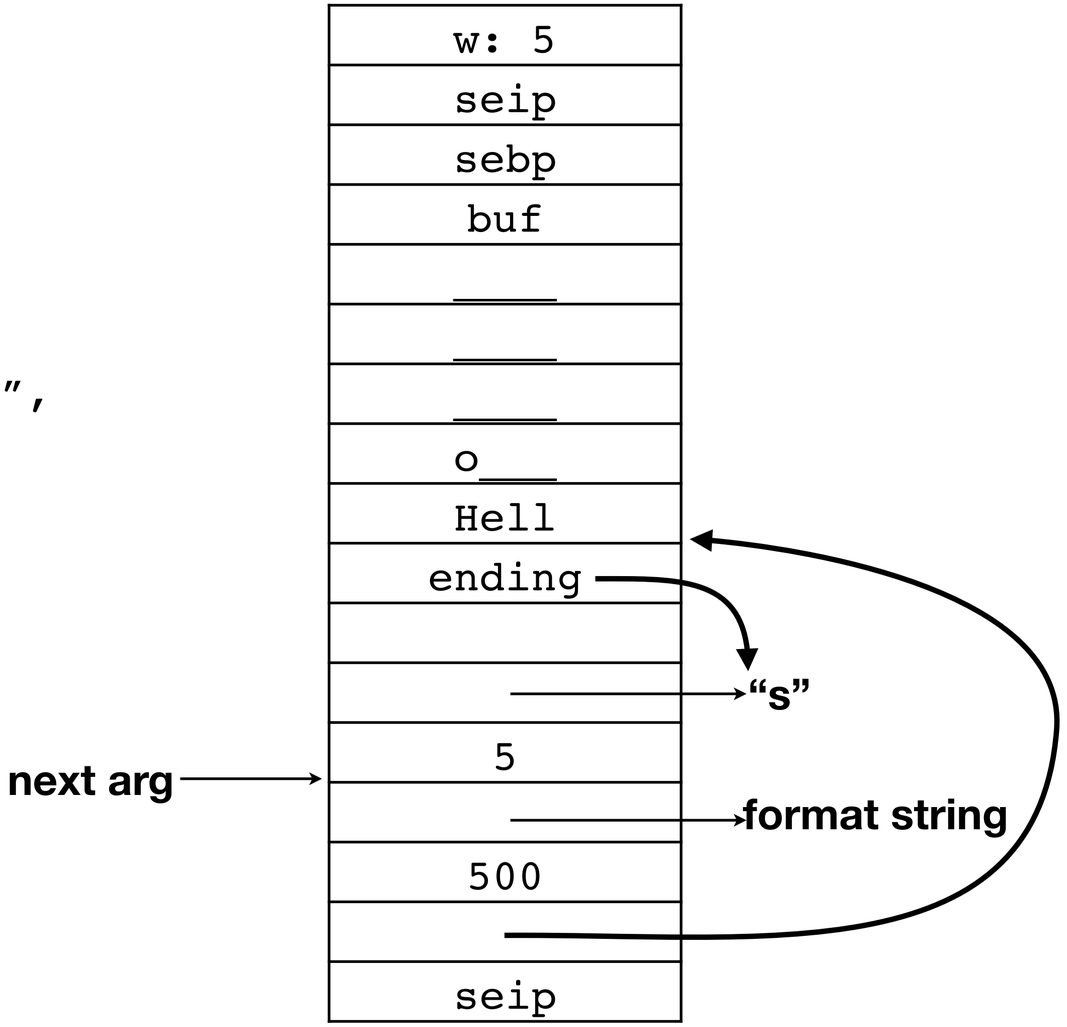
The way snprintf() normally works

```
void foo(int w) {
  char buf[500];
  const char *ending = w==1? “”:”s”;
  snprintf(buf, 500, “Hello %d world%s”,
         w, ending);
}
…
foo(5);
```



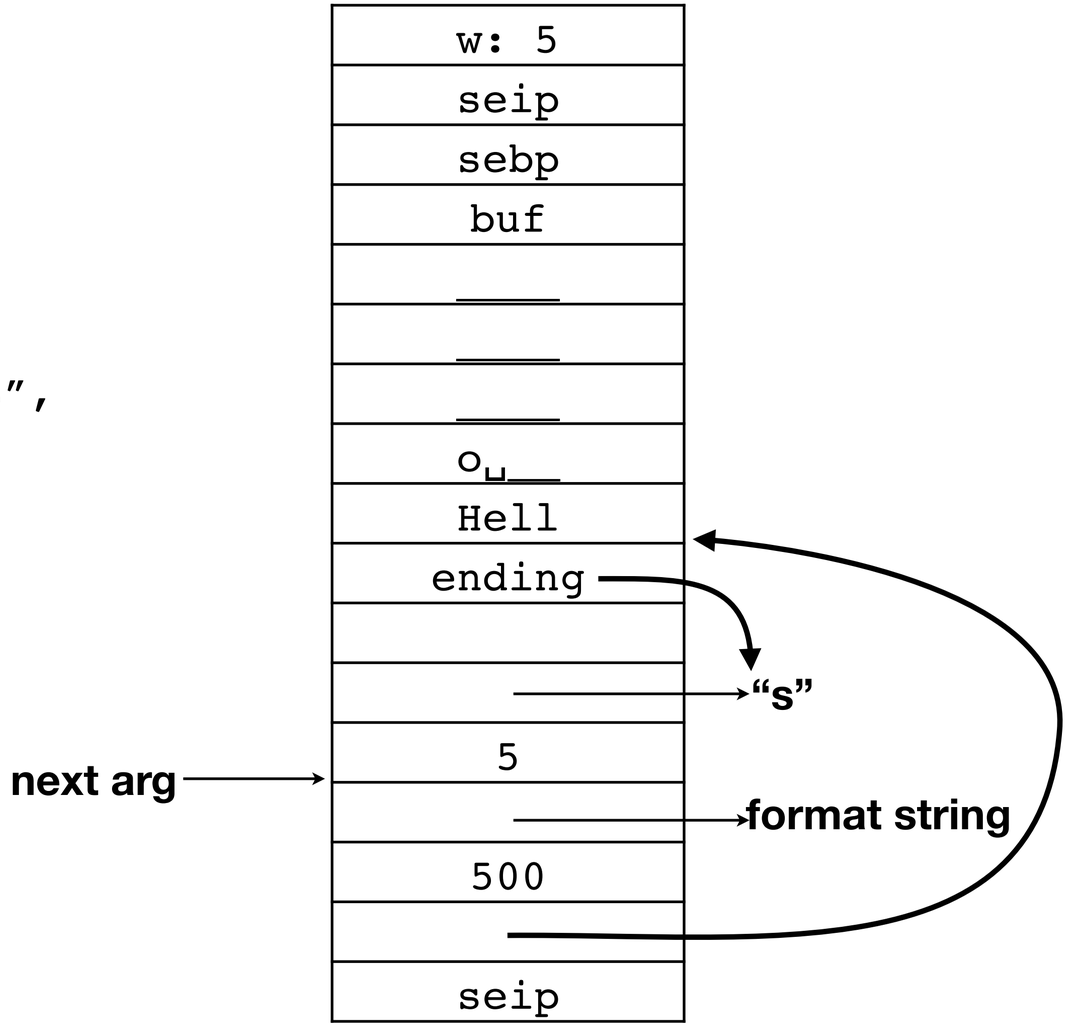
The way snprintf() normally works

```
void foo(int w) {
  char buf[500];
  const char *ending = w==1? “”:”s”;
  snprintf(buf, 500, “Hello␣%d world%s”,
         w, ending);
}
…
foo(5);
```



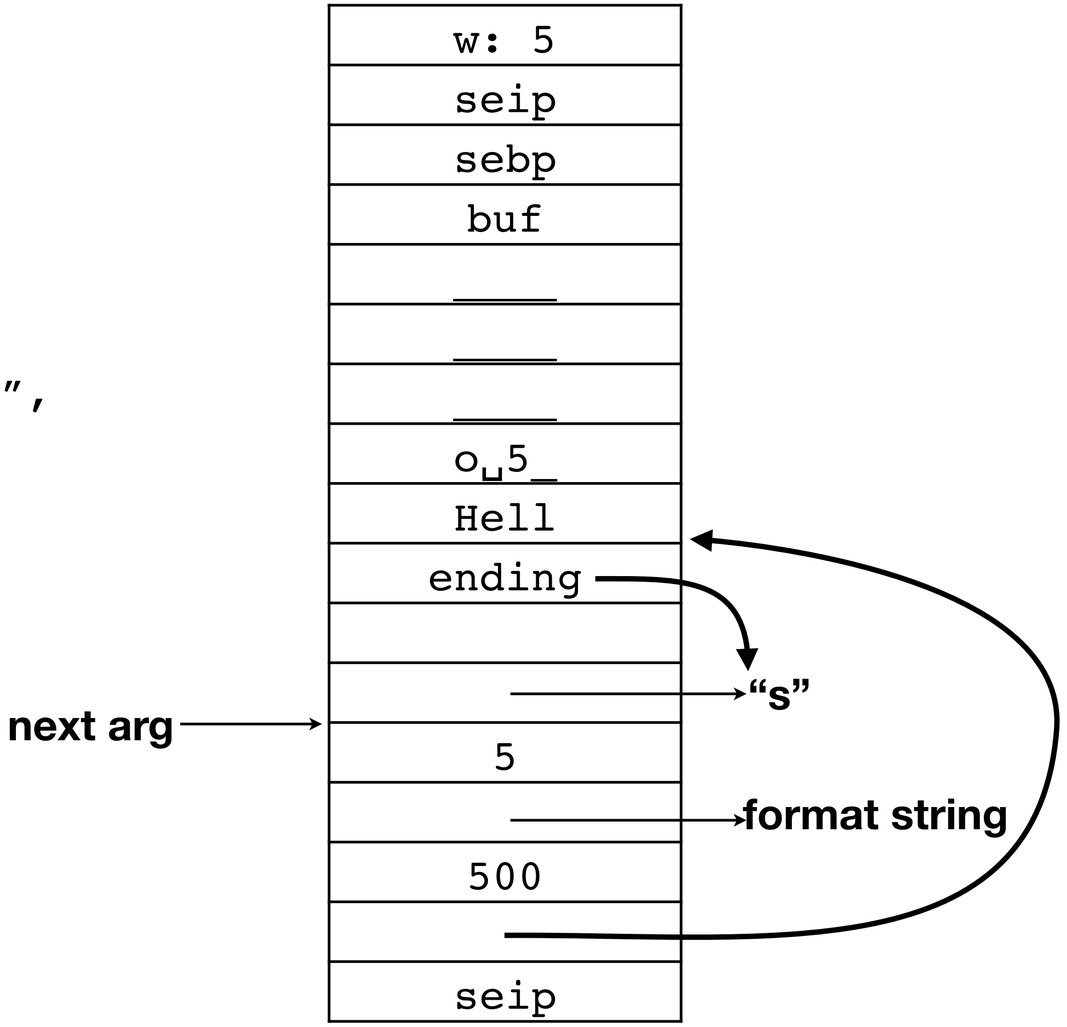
The way snprintf() normally works

```
void foo(int w) {
  char buf[500];
  const char *ending = w==1? “”:”s”;
  snprintf(buf, 500, “Hello %d world%s”,
         w, ending);
}
…
foo(5);
```



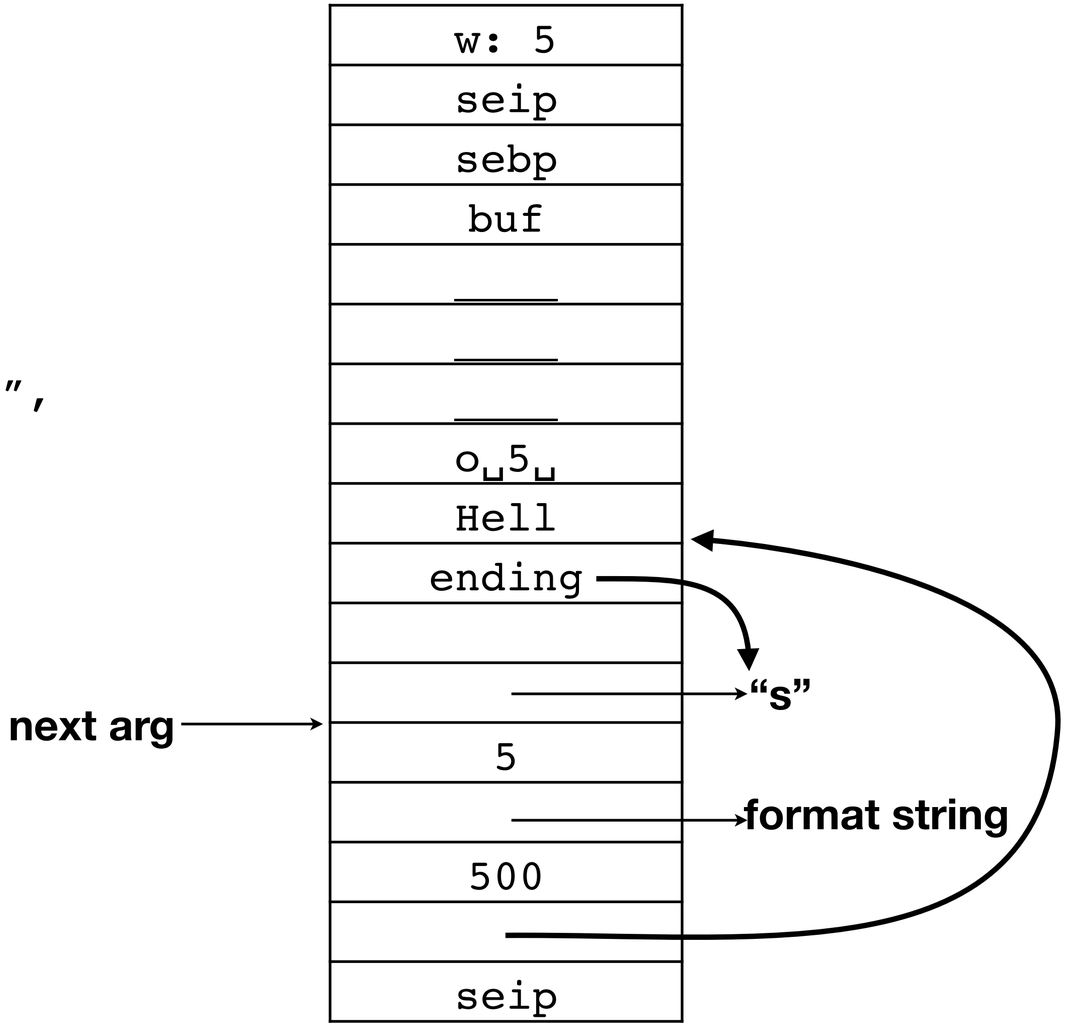
The way snprintf() normally works

```
void foo(int w) {
  char buf[500];
  const char *ending = w==1? “”:”s”;
  snprintf(buf, 500, “Hello %d␣world%s”,
         w, ending);
}
…
foo(5);
```



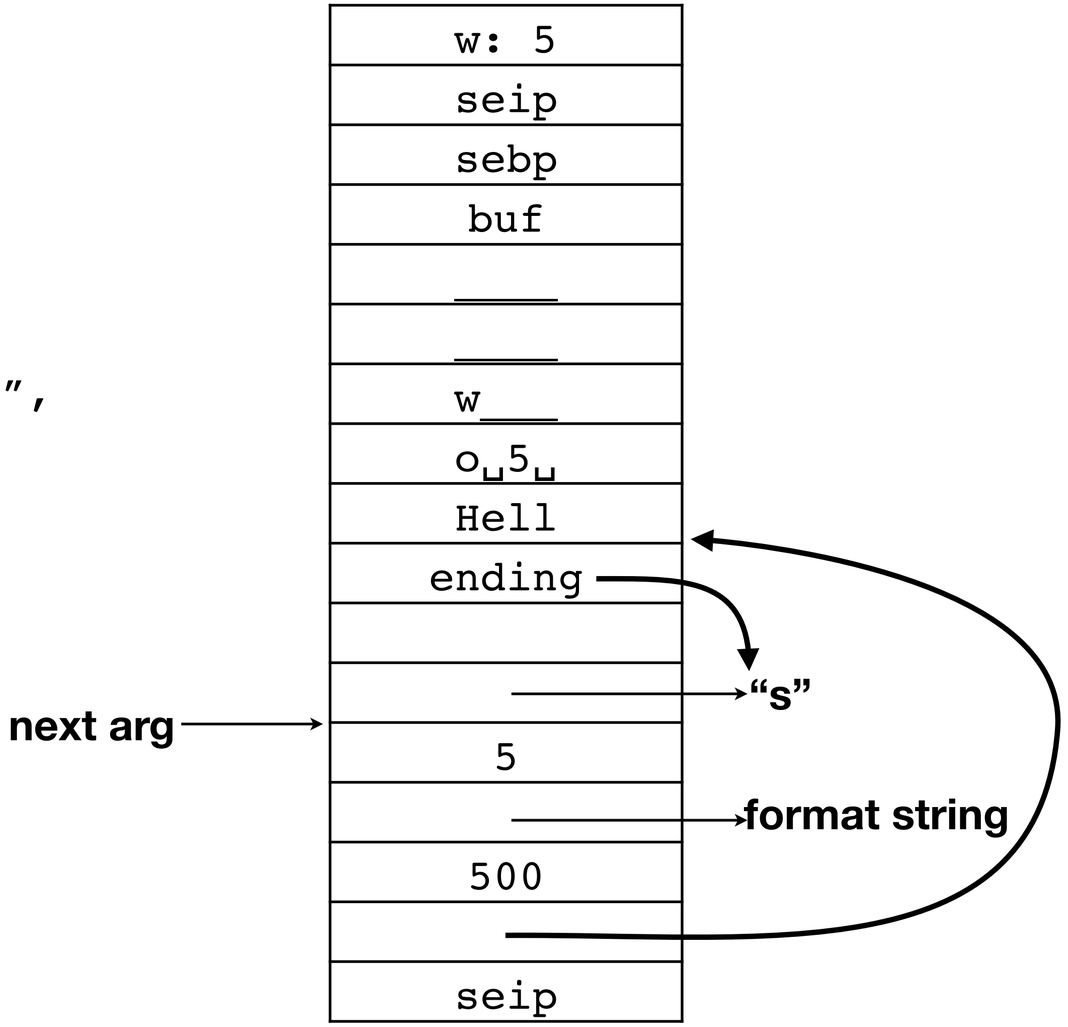
The way snprintf() normally works

```
void foo(int w) {
  char buf[500];
  const char *ending = w==1? “”:”s”;
  snprintf(buf, 500, “Hello %d world%s”,
         w, ending);
}
…
foo(5);
```



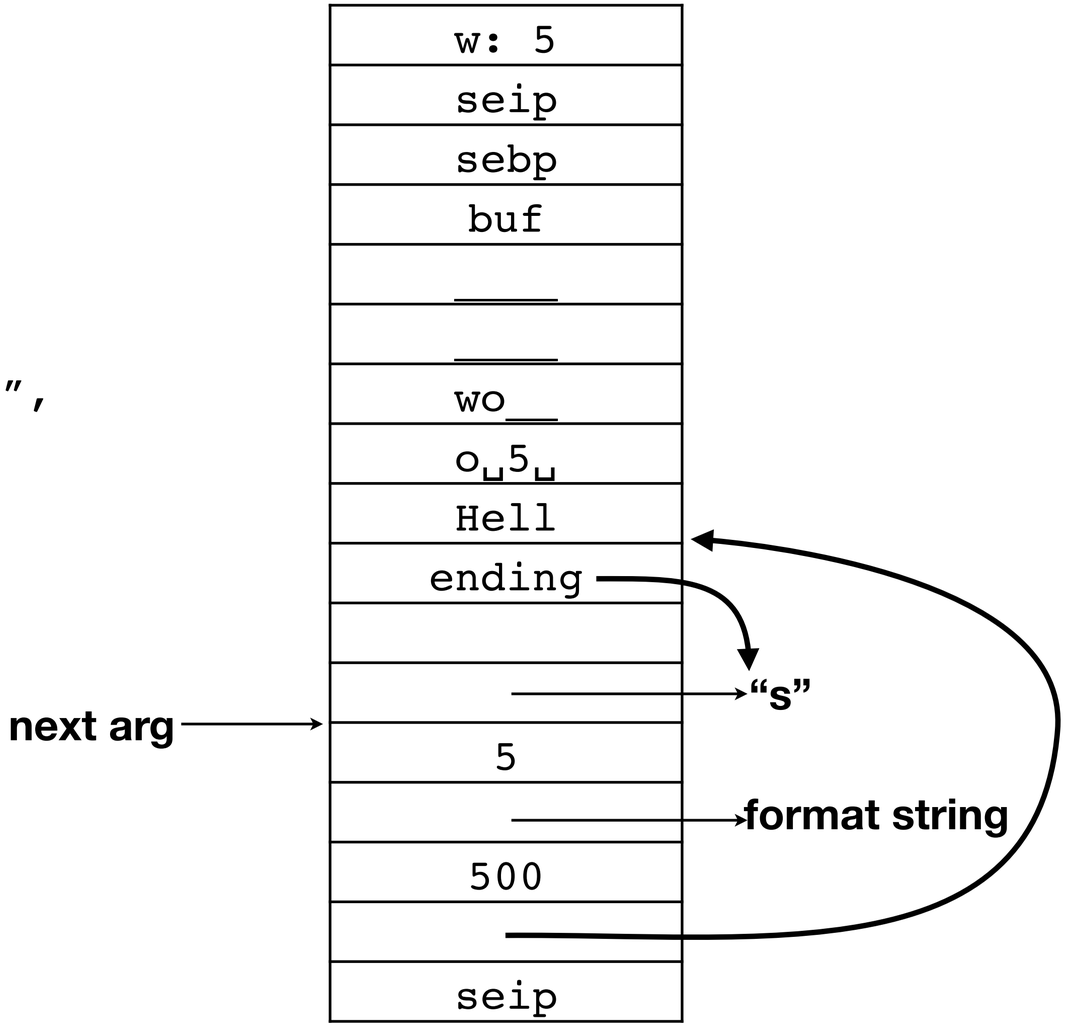
The way snprintf() normally works

```
void foo(int w) {
  char buf[500];
  const char *ending = w==1? “”:”s”;
  snprintf(buf, 500, “Hello %d world%s”,
         w, ending);
}
…
foo(5);
```



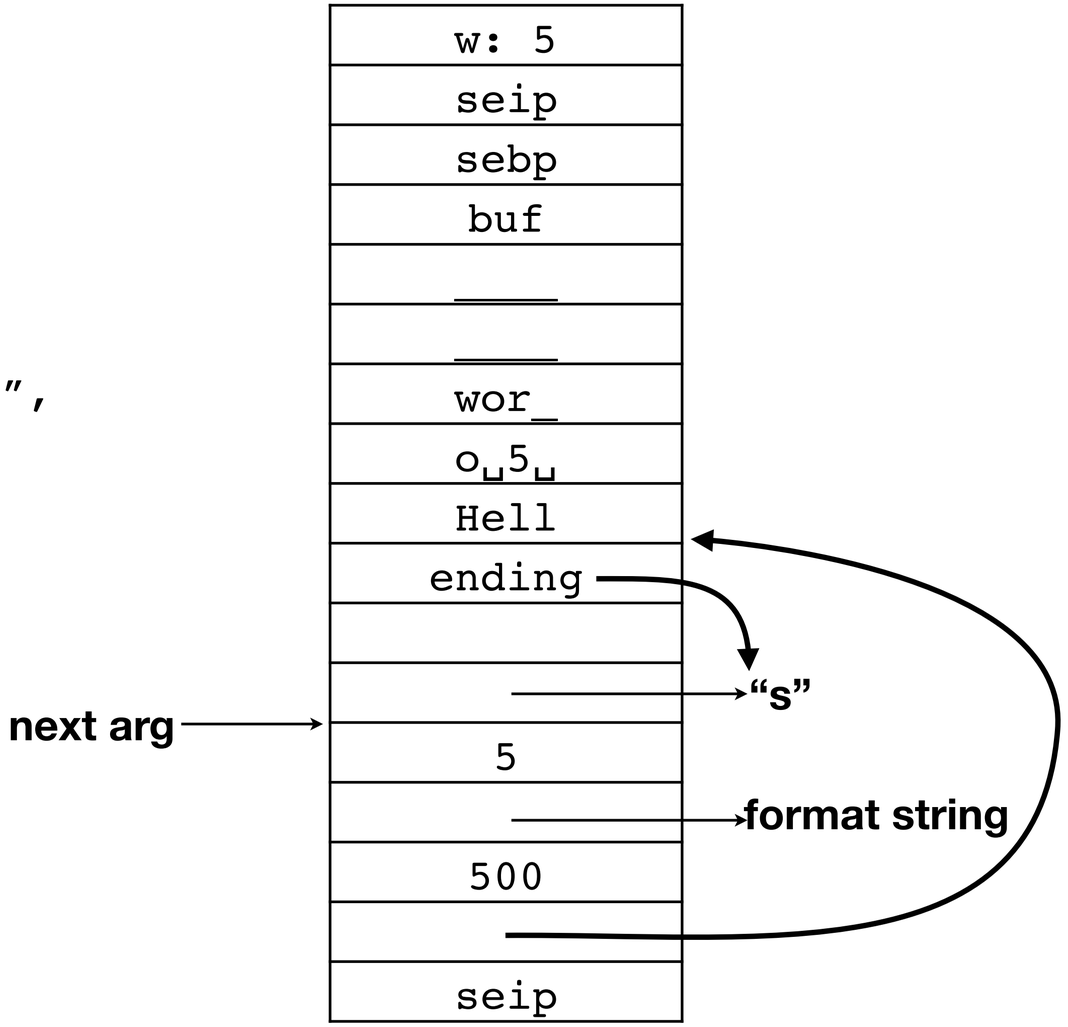
The way snprintf() normally works

```
void foo(int w) {
  char buf[500];
  const char *ending = w==1? “”:”s”;
  snprintf(buf, 500, “Hello %d world%s”,
         w, ending);
}
…
foo(5);
```



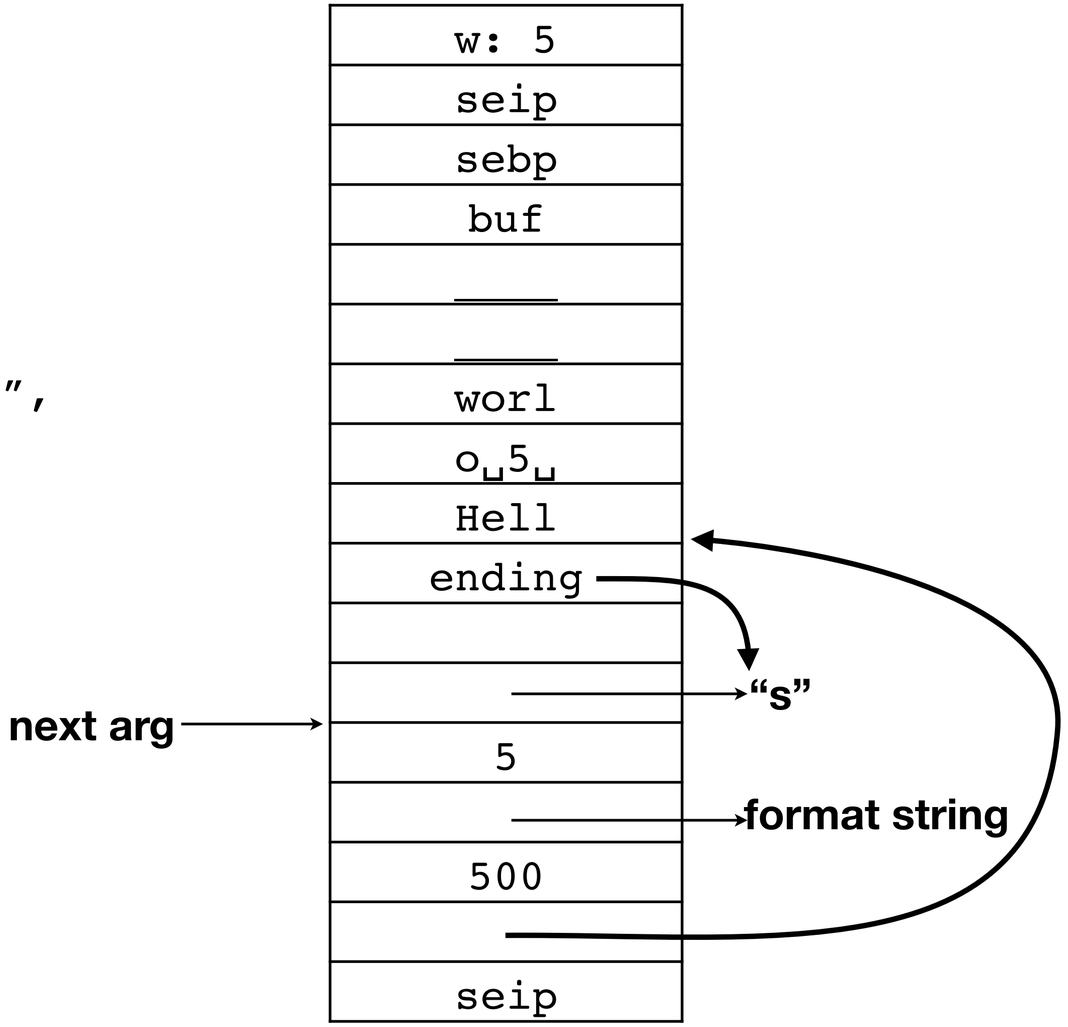
The way snprintf() normally works

```
void foo(int w) {
  char buf[500];
  const char *ending = w==1? “”:”s”;
  snprintf(buf, 500, “Hello %d world%s”,
         w, ending);
}
…
foo(5);
```



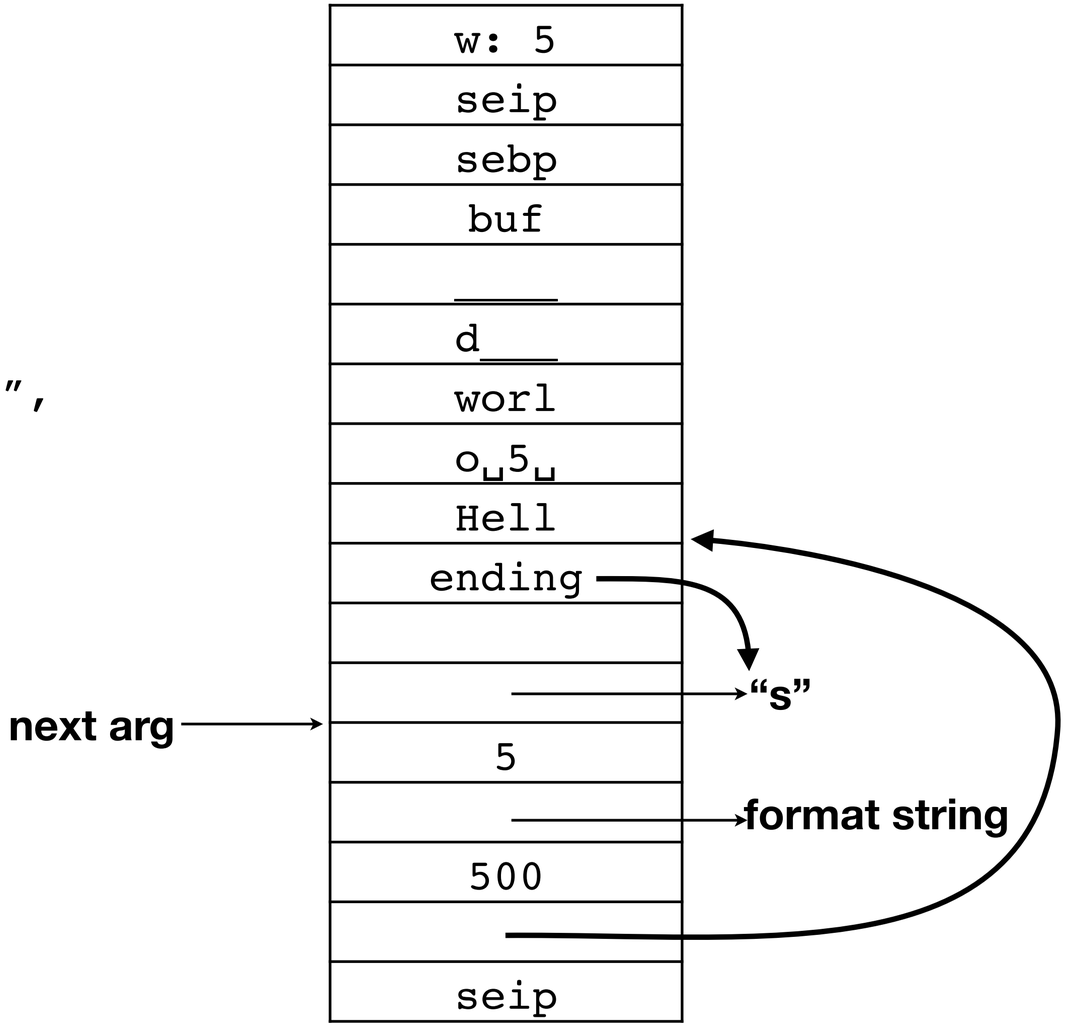
The way snprintf() normally works

```
void foo(int w) {
  char buf[500];
  const char *ending = w==1? “”:”s”;
  snprintf(buf, 500, “Hello %d world%s”,
         w, ending);
}
…
foo(5);
```



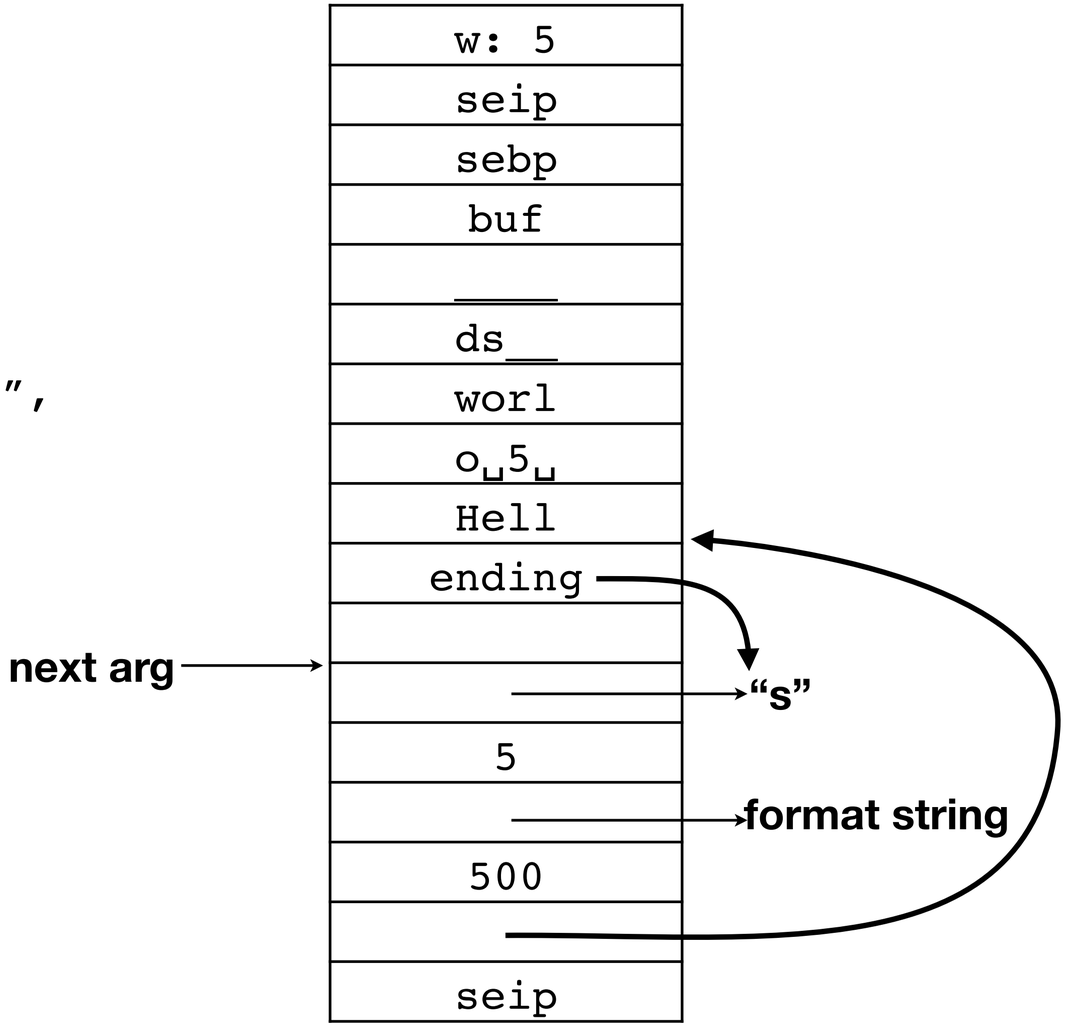
The way snprintf() normally works

```
void foo(int w) {
  char buf[500];
  const char *ending = w==1? “”:”s”;
  snprintf(buf, 500, “Hello %d world%s”,
         w, ending);
}
…
foo(5);
```



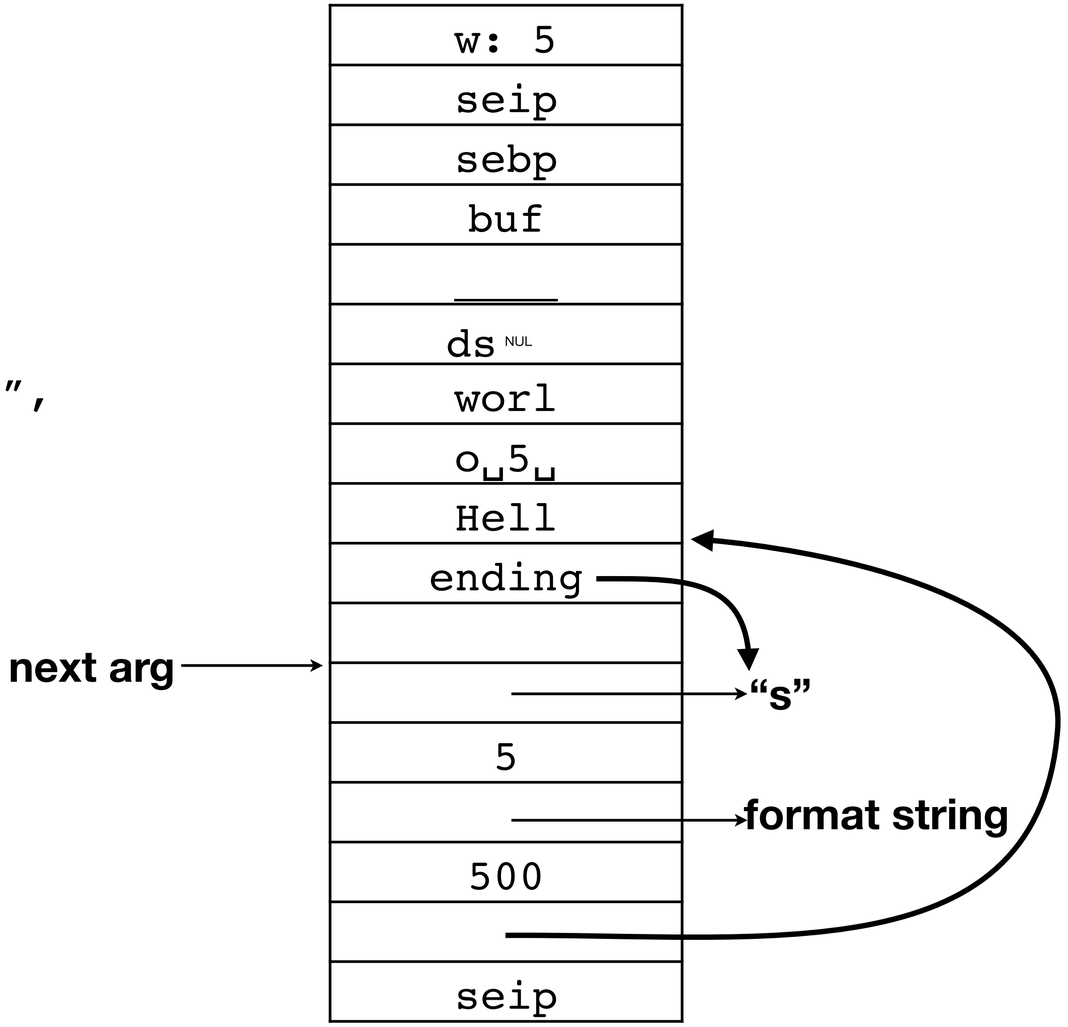
The way snprintf() normally works

```
void foo(int w) {
  char buf[500];
  const char *ending = w==1? “”:”s”;
  snprintf(buf, 500, “Hello %d world%s”,
         w, ending);
}
…
foo(5);
```



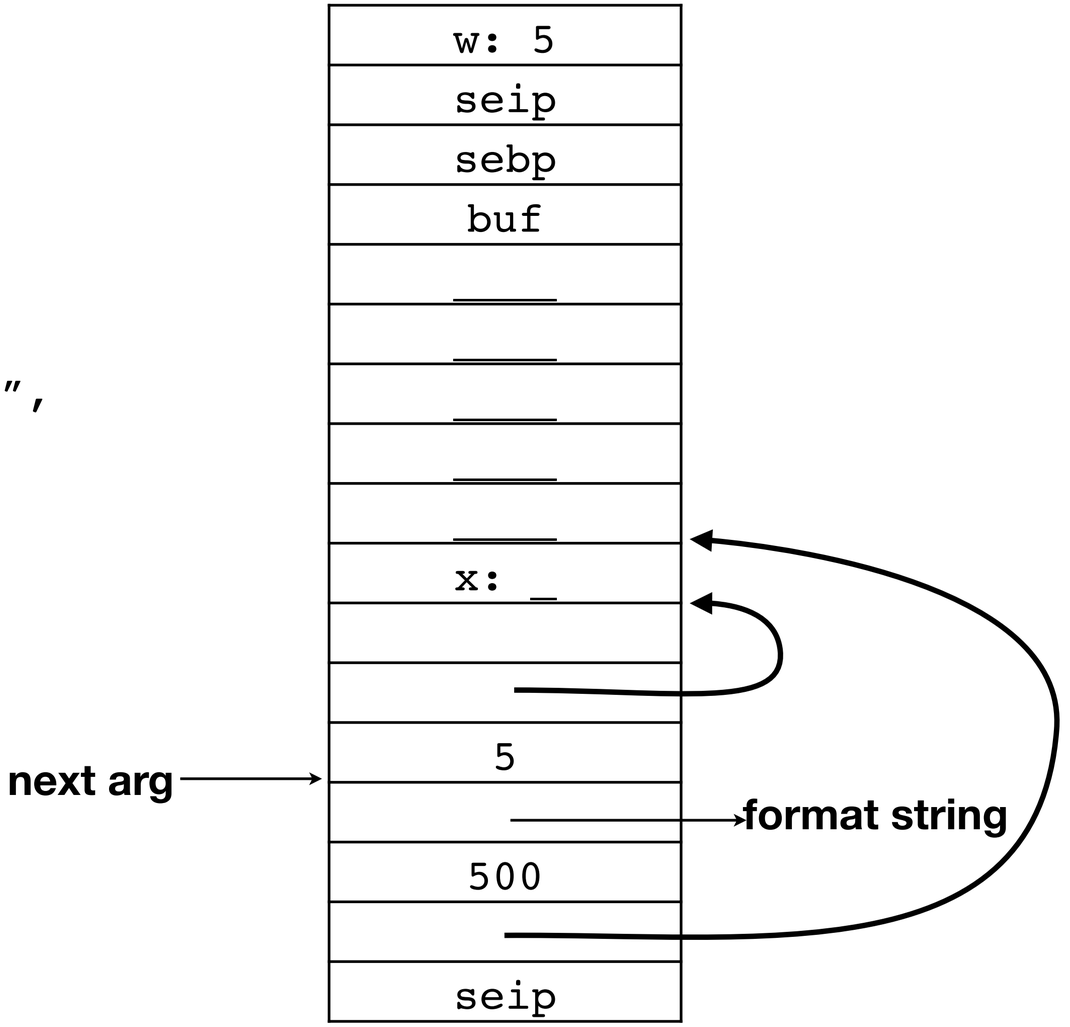
The way snprintf() normally works

```
void foo(int w) {
  char buf[500];
  const char *ending = w==1? “”:”s”;
  snprintf(buf, 500, “Hello %d world%s”,
         w, ending);
}
…
foo(5);
```



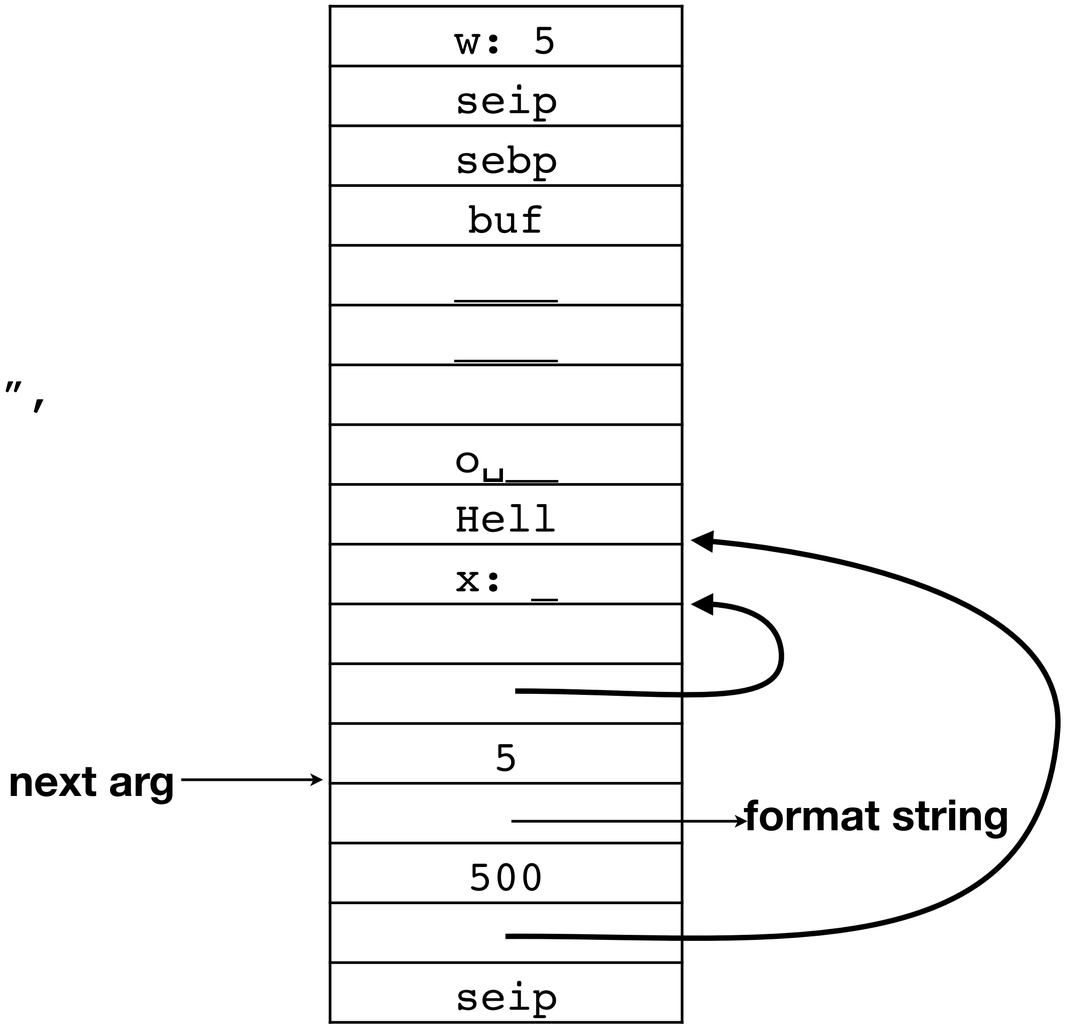
Now with %n

```
void foo(int w) {
  char buf[500];
  int x;
  snprintf(buf, 500, “Hello␣%d world%n”,
          w, &x);
}
…
foo(5);
```



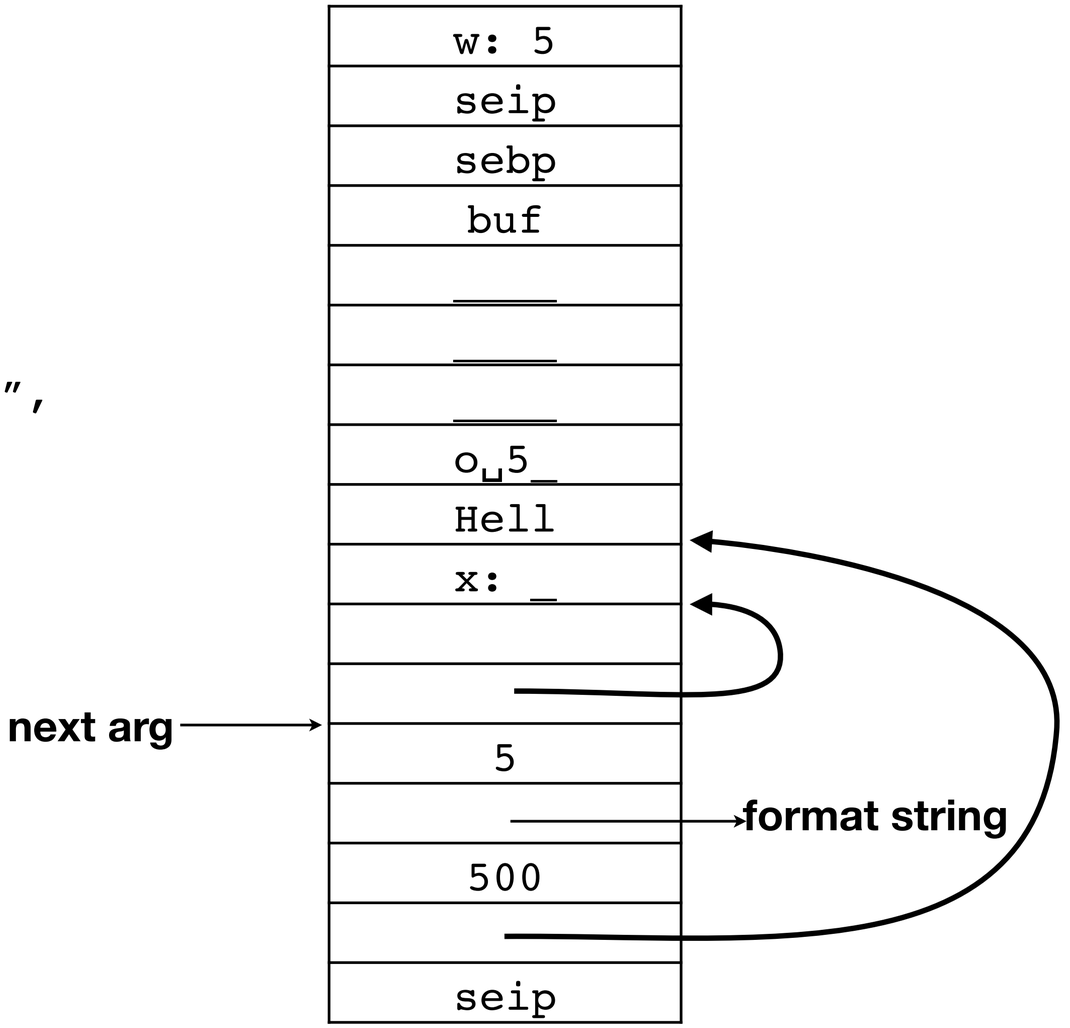
Now with %n

```
void foo(int w) {
  char buf[500];
  int x;
  snprintf(buf, 500, “Hello %d world%n”,
          w, &x);
}
…
foo(5);
```



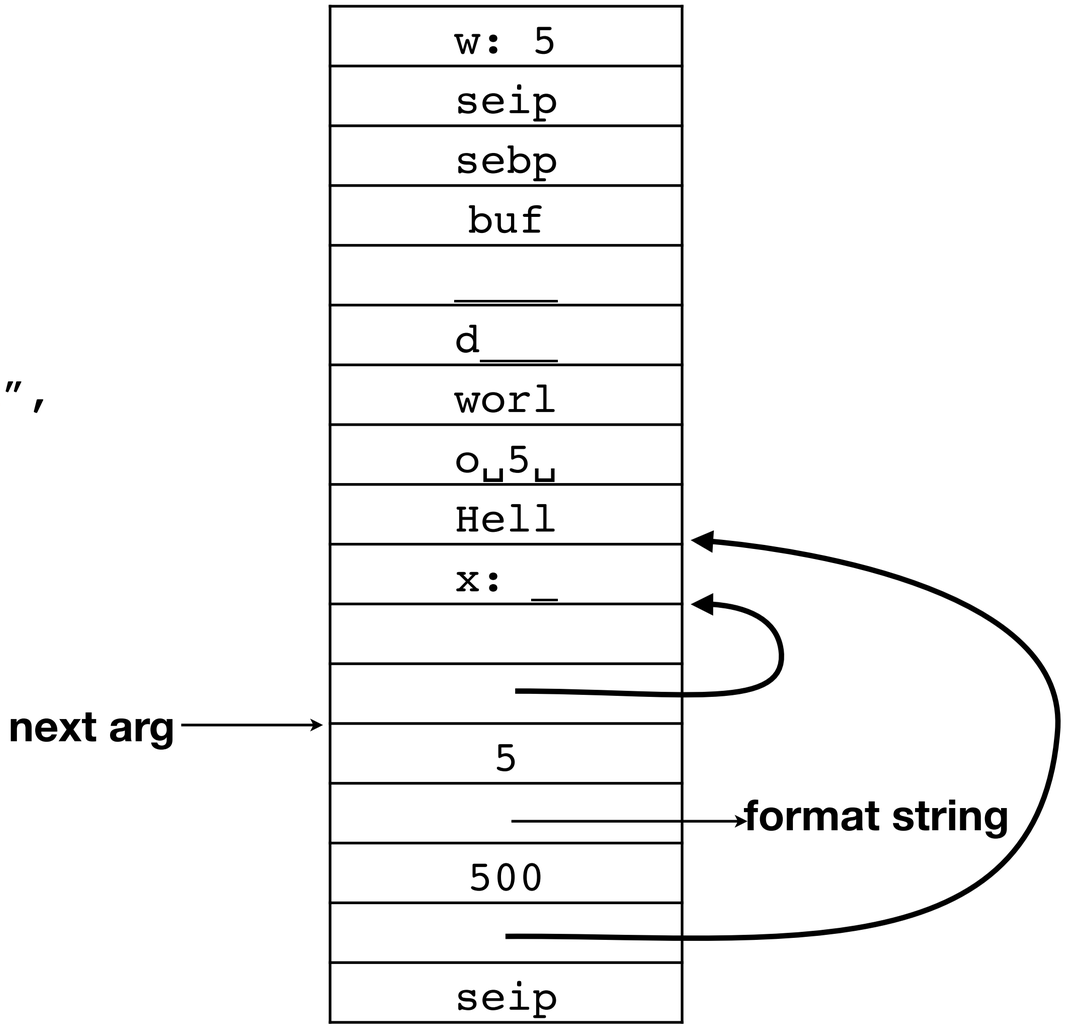
Now with %n

```
void foo(int w) {
  char buf[500];
  int x;
  snprintf(buf, 500, “Hello %d␣world%n”,
          w, &x);
}
…
foo(5);
```



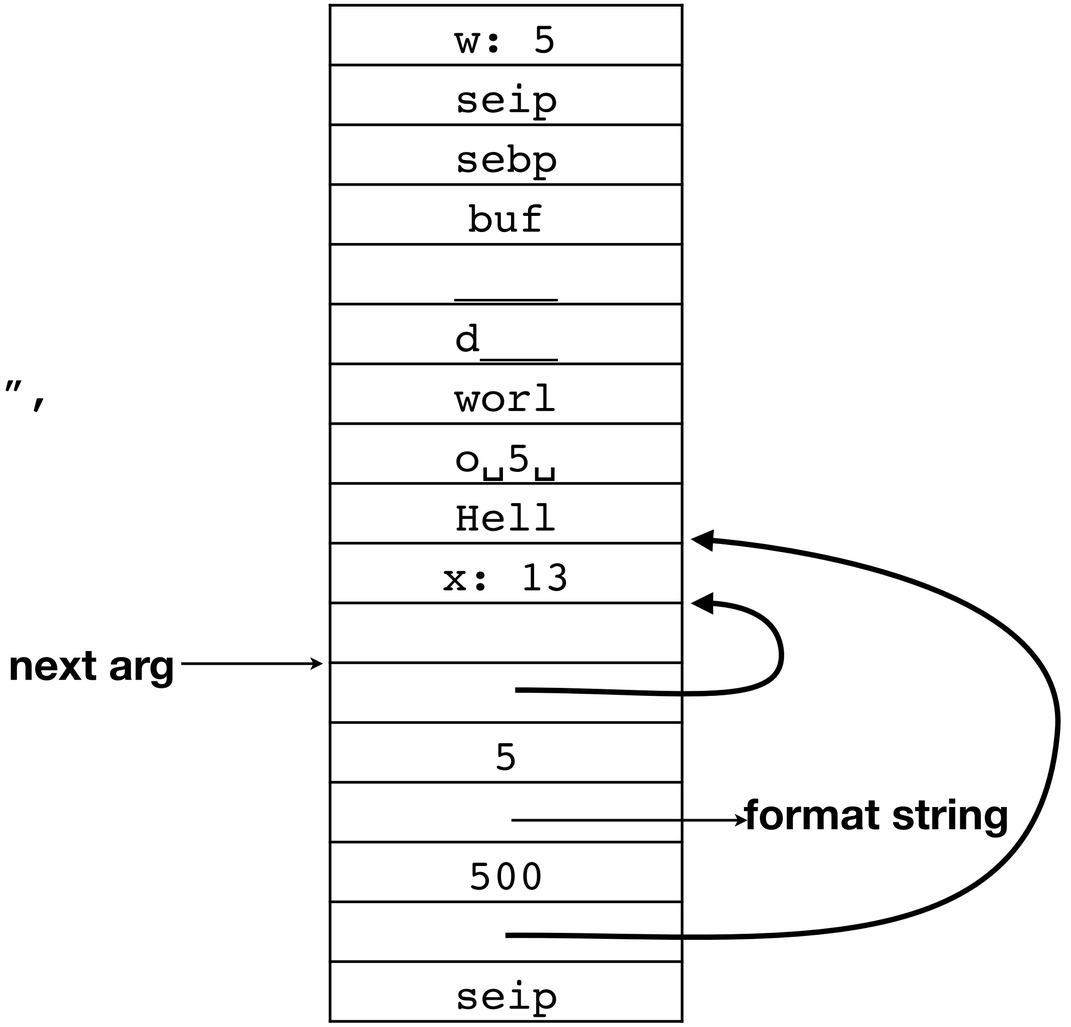
Now with %n

```
void foo(int w) {
  char buf[500];
  int x;
  snprintf(buf, 500, “Hello %d world%n”,
         w, &x);
}
…
foo(5);
```



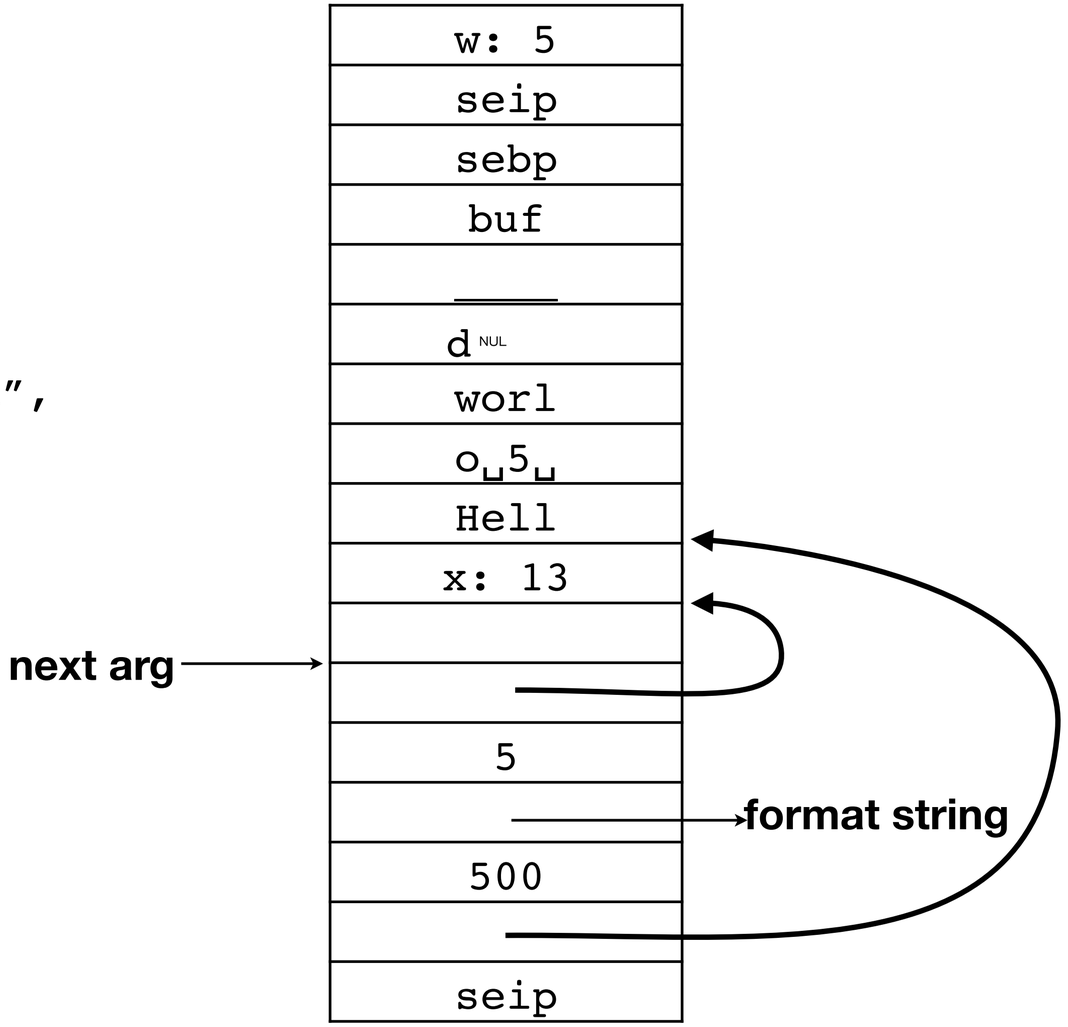
Now with %n

```
void foo(int w) {
  char buf[500];
  int x;
  snprintf(buf, 500, “Hello %d world%n”,
         w, &x);
}
…
foo(5);
```



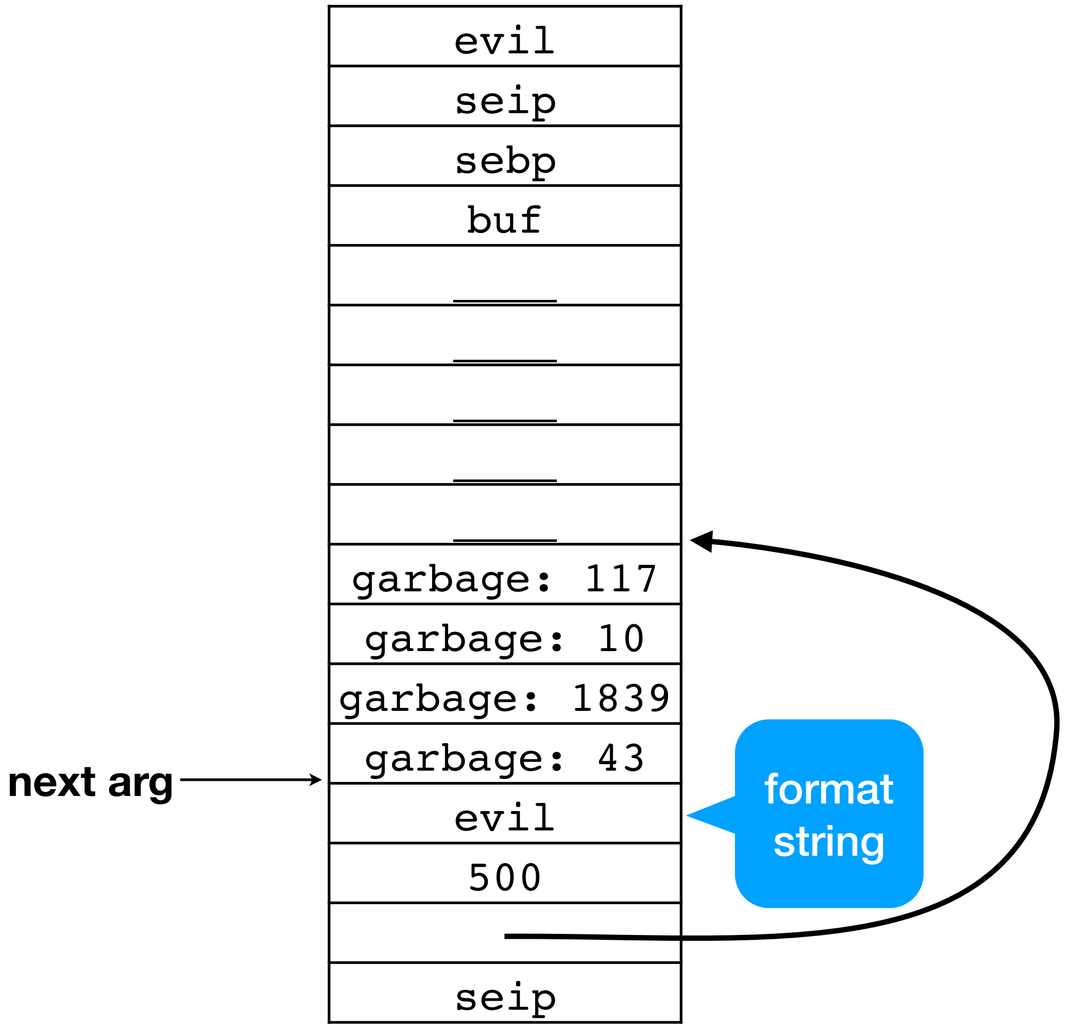
Now with %n

```
void foo(int w) {
  char buf[500];
  int x;
  snprintf(buf, 500, “Hello %d world%n”,
         w, &x);
}
…
foo(5);
```



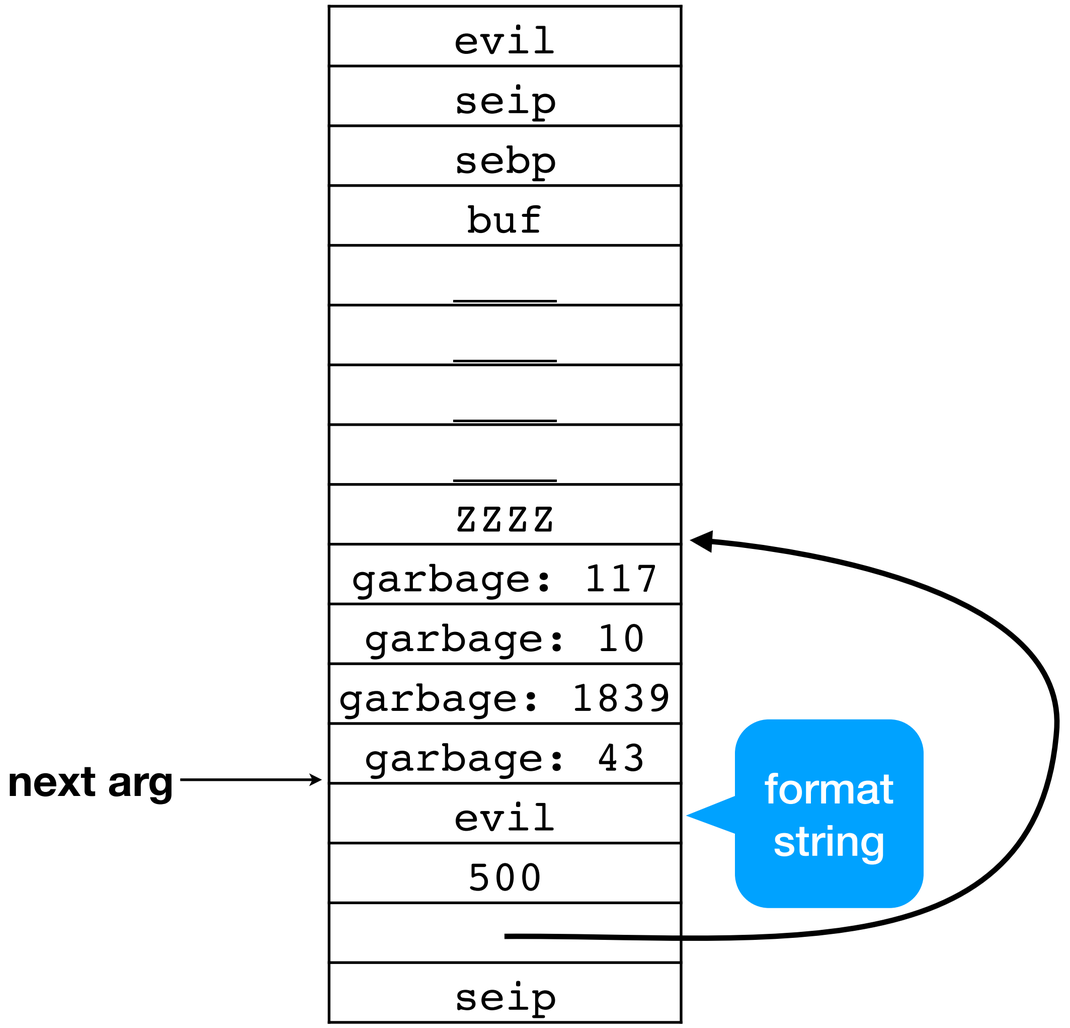
Attacker controlled format string

```
void foo(const char *evil) {
  char buf[500];
  snprintf(buf, 500, evil);
}
…
foo("ZZZZ%x%x%x%x%x");
```



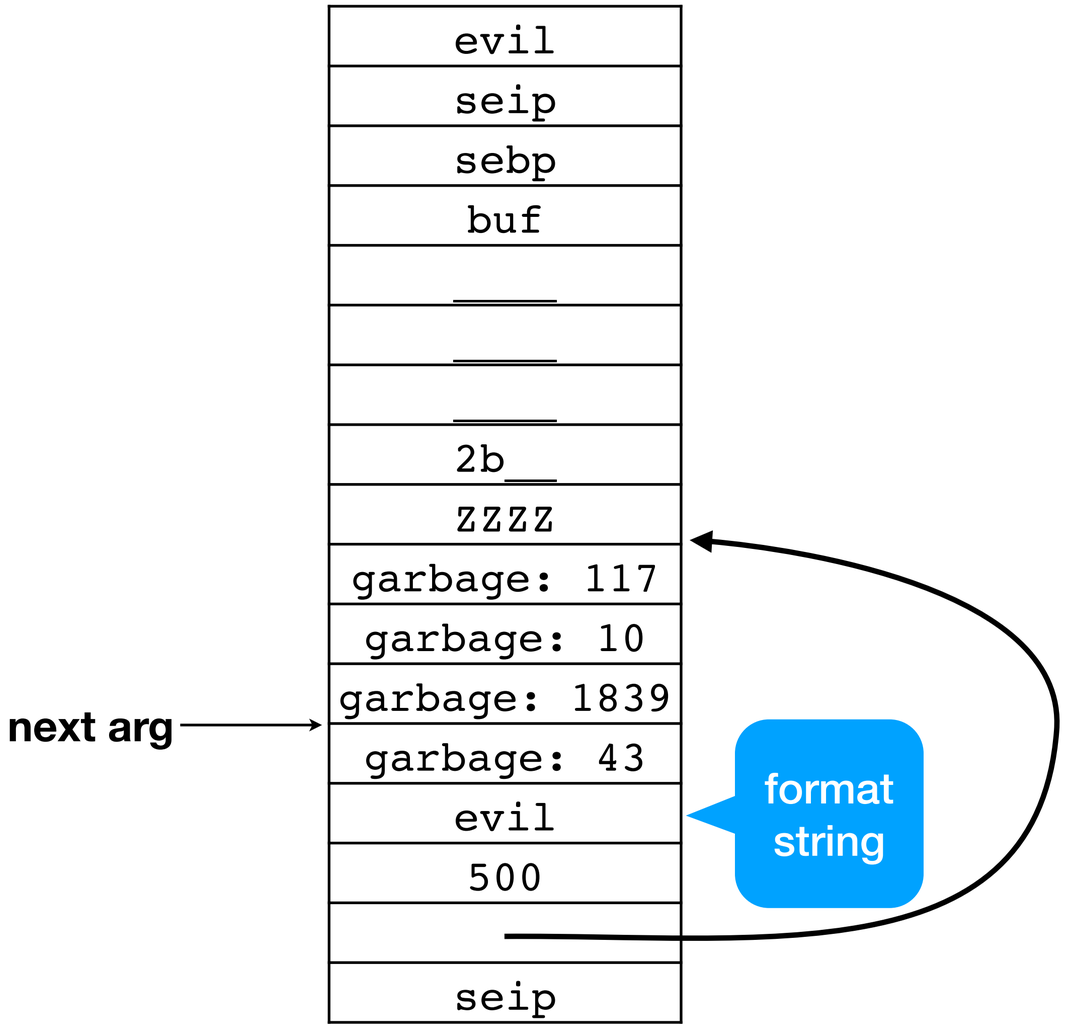
Attacker controlled format string

```
void foo(const char *evil) {
  char buf[500];
  snprintf(buf, 500, evil);
}
…
foo("ZZZZ%x%x%x%x%x");
```



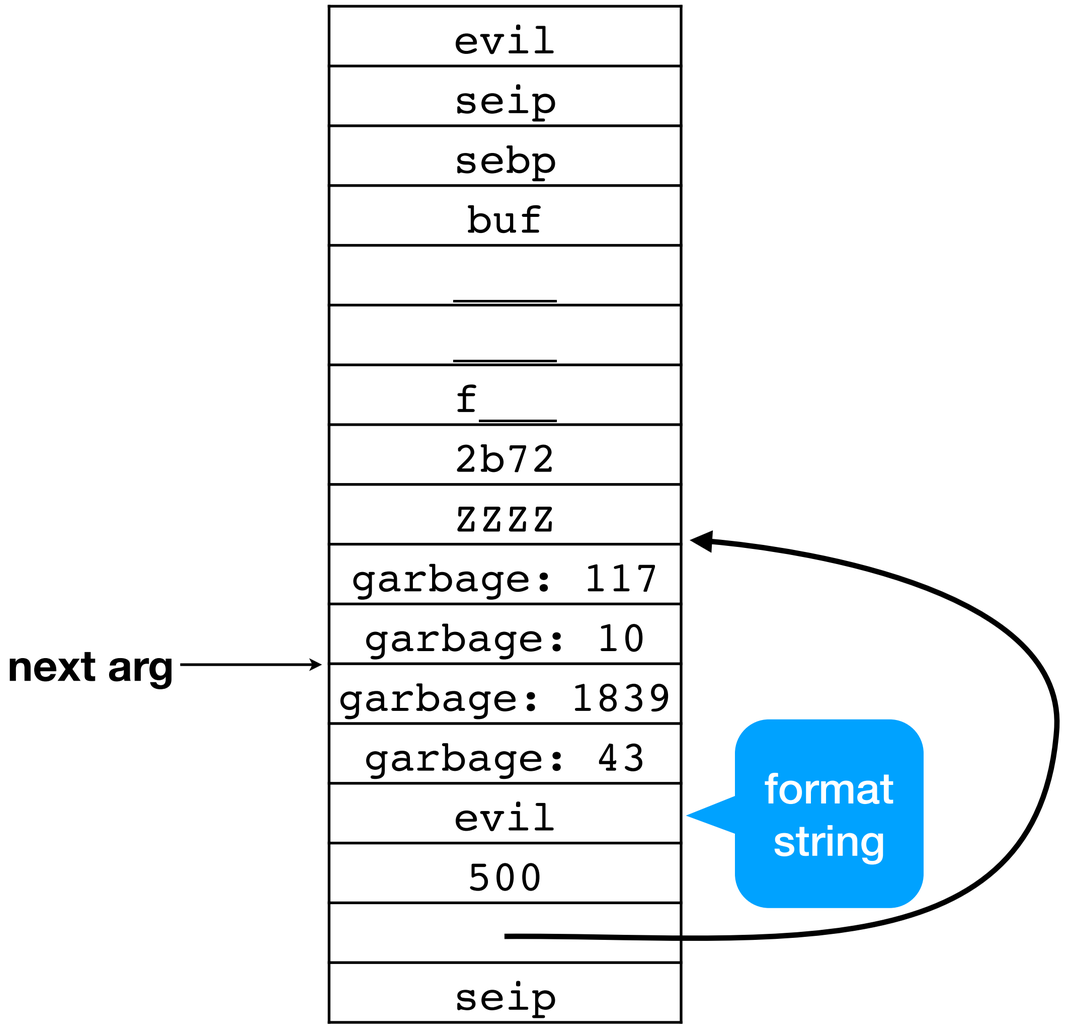
Attacker controlled format string

```
void foo(const char *evil) {
  char buf[500];
  snprintf(buf, 500, evil);
}
…
foo("ZZZZ%x%x%x%x%x");
```



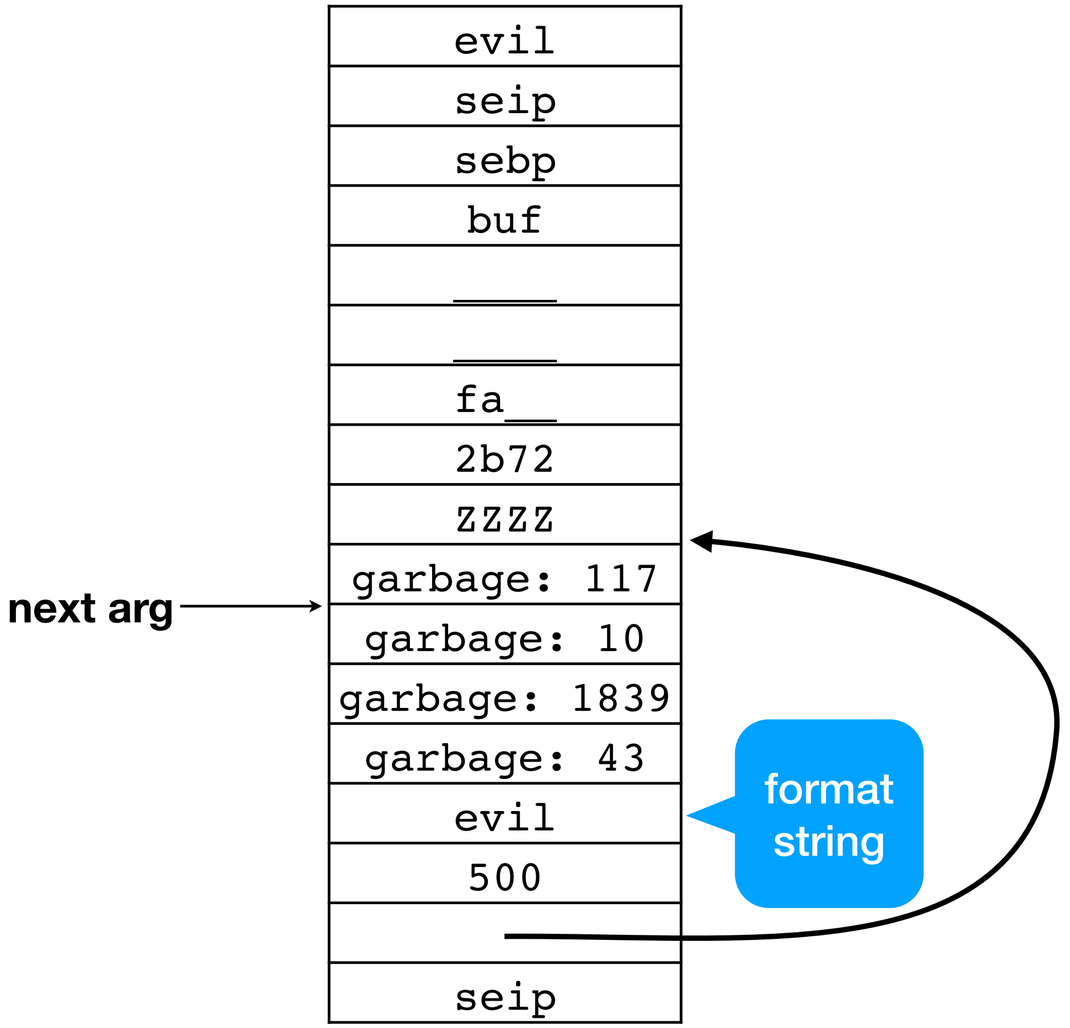
Attacker controlled format string

```
void foo(const char *evil) {
  char buf[500];
  snprintf(buf, 500, evil);
}
…
foo("ZZZZ%x%x%x%x%x");
```



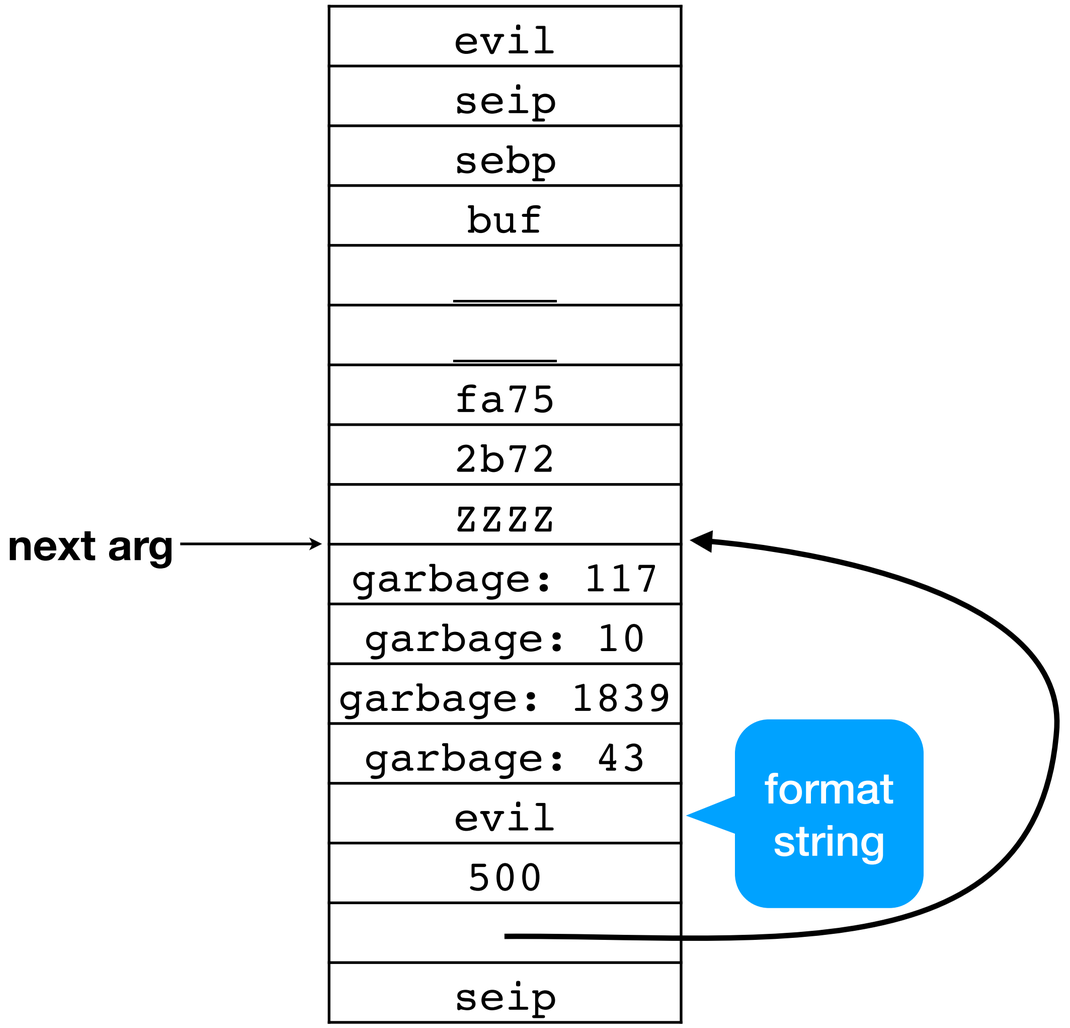
Attacker controlled format string

```
void foo(const char *evil) {
  char buf[500];
  snprintf(buf, 500, evil);
}
…
foo(“ZZZZ%x%x%x%x%x”);
```



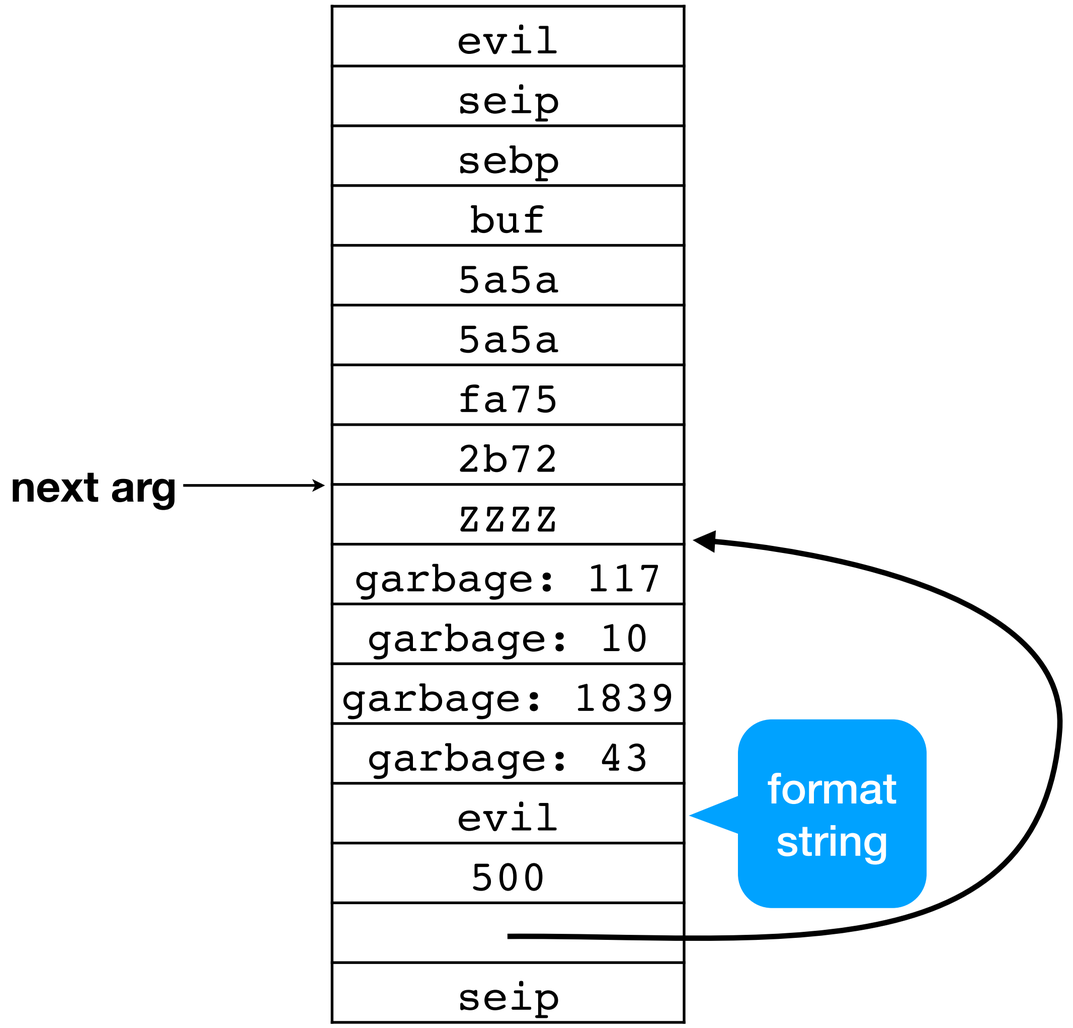
Attacker controlled format string

```
void foo(const char *evil) {
  char buf[500];
  snprintf(buf, 500, evil);
}
…
foo("ZZZZ%x%x%x%x%x");
```



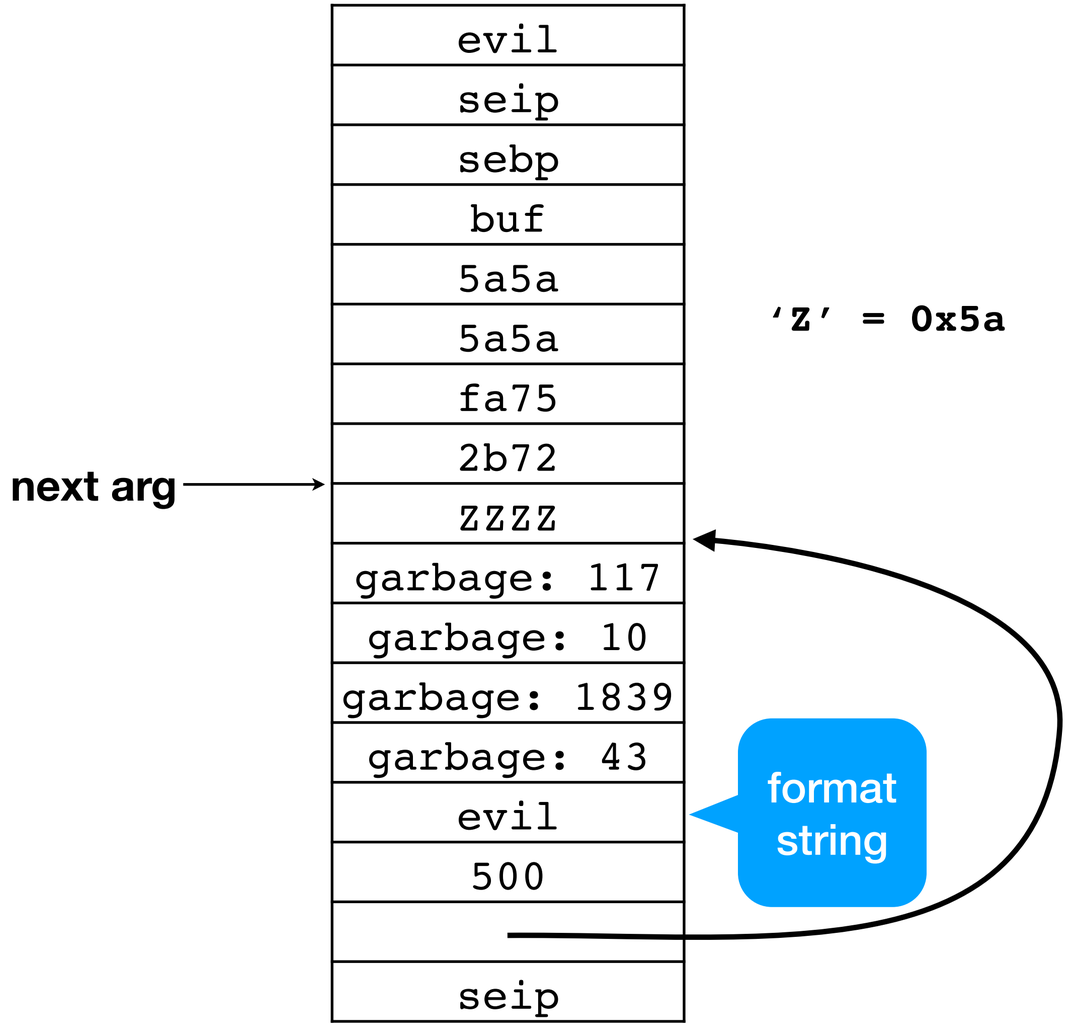
Attacker controlled format string

```
void foo(const char *evil) {
  char buf[500];
  snprintf(buf, 500, evil);
}
…
foo("ZZZZ%x%x%x%x%x");
```



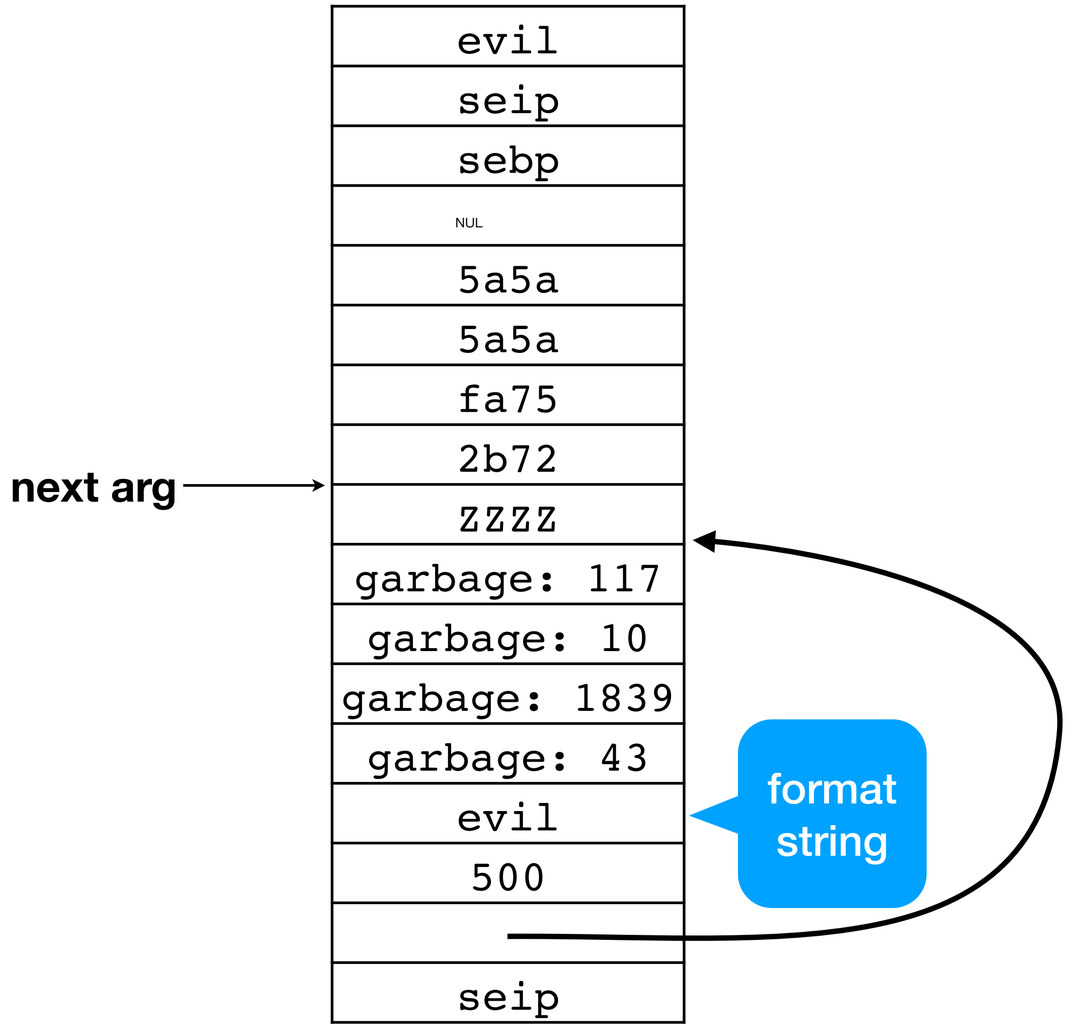
Attacker controlled format string

```
void foo(const char *evil) {
  char buf[500];
  snprintf(buf, 500, evil);
}
…
foo(“ZZZZ%x%x%x%x%x”);
```



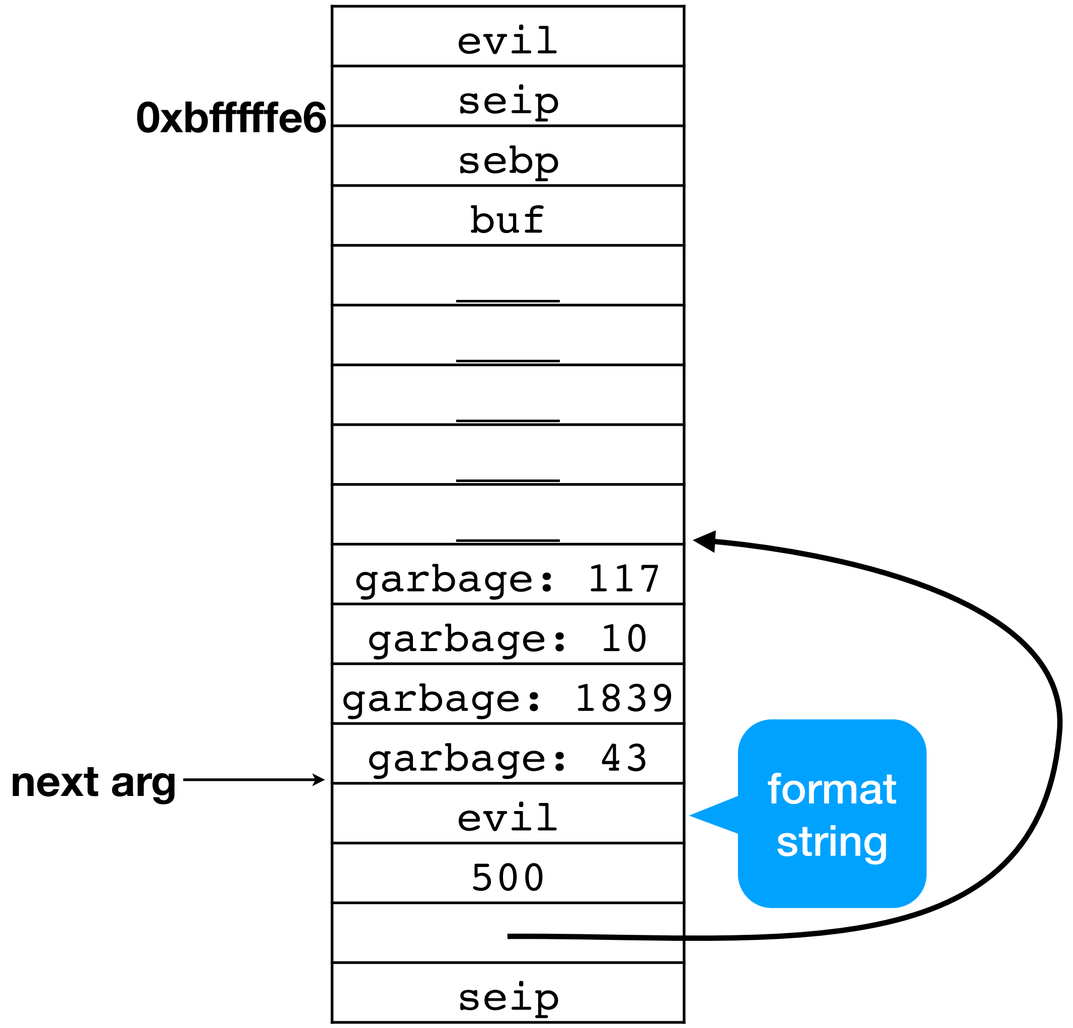
Attacker controlled format string

```
void foo(const char *evil) {
  char buf[500];
  snprintf(buf, 500, evil);
}
…
foo("ZZZZ%x%x%x%x%x");
```



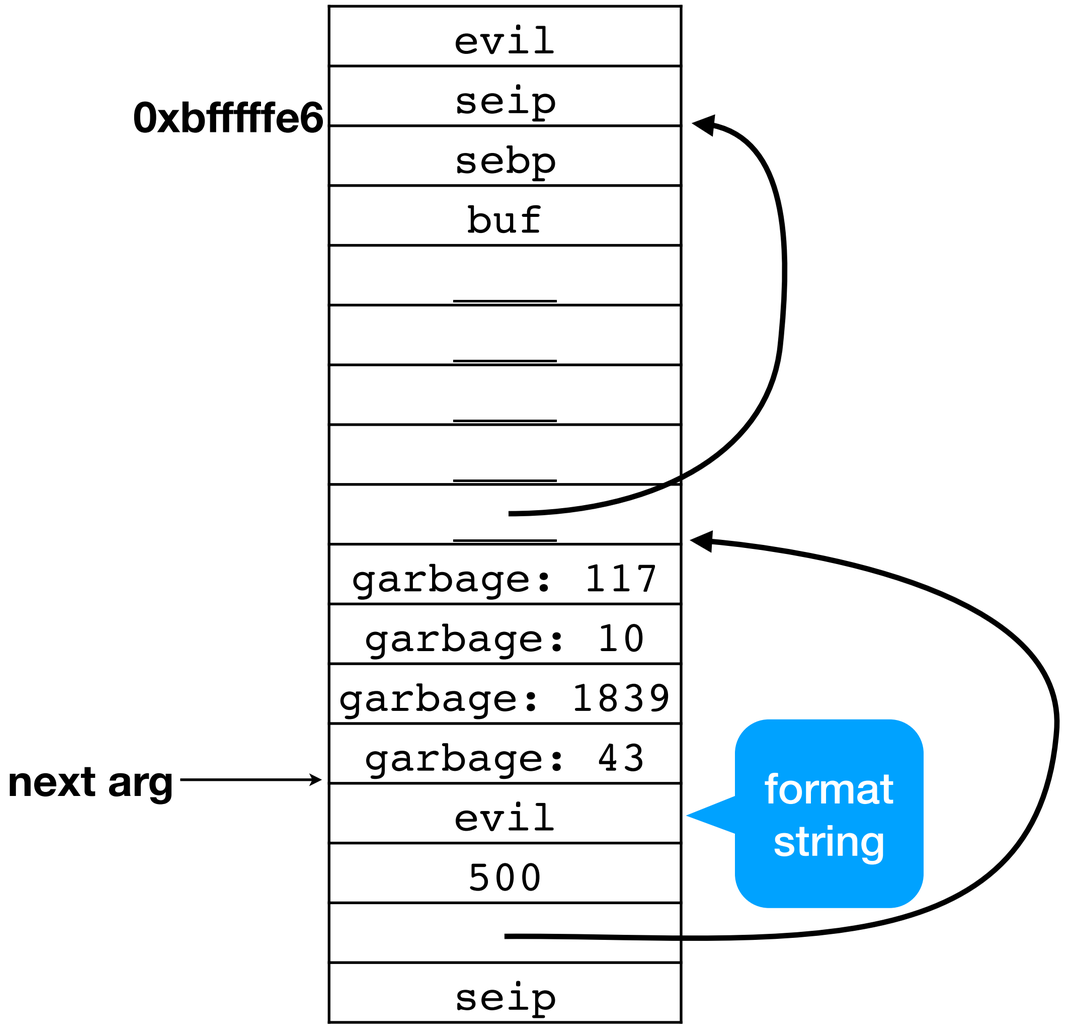
Overwriting seip

```
void foo(const char *evil) {
  char buf[500];
  snprintf(buf, 500, evil);
}
…
foo("\xe6\xff\xff\xbf%x%x%x%x%n");
```



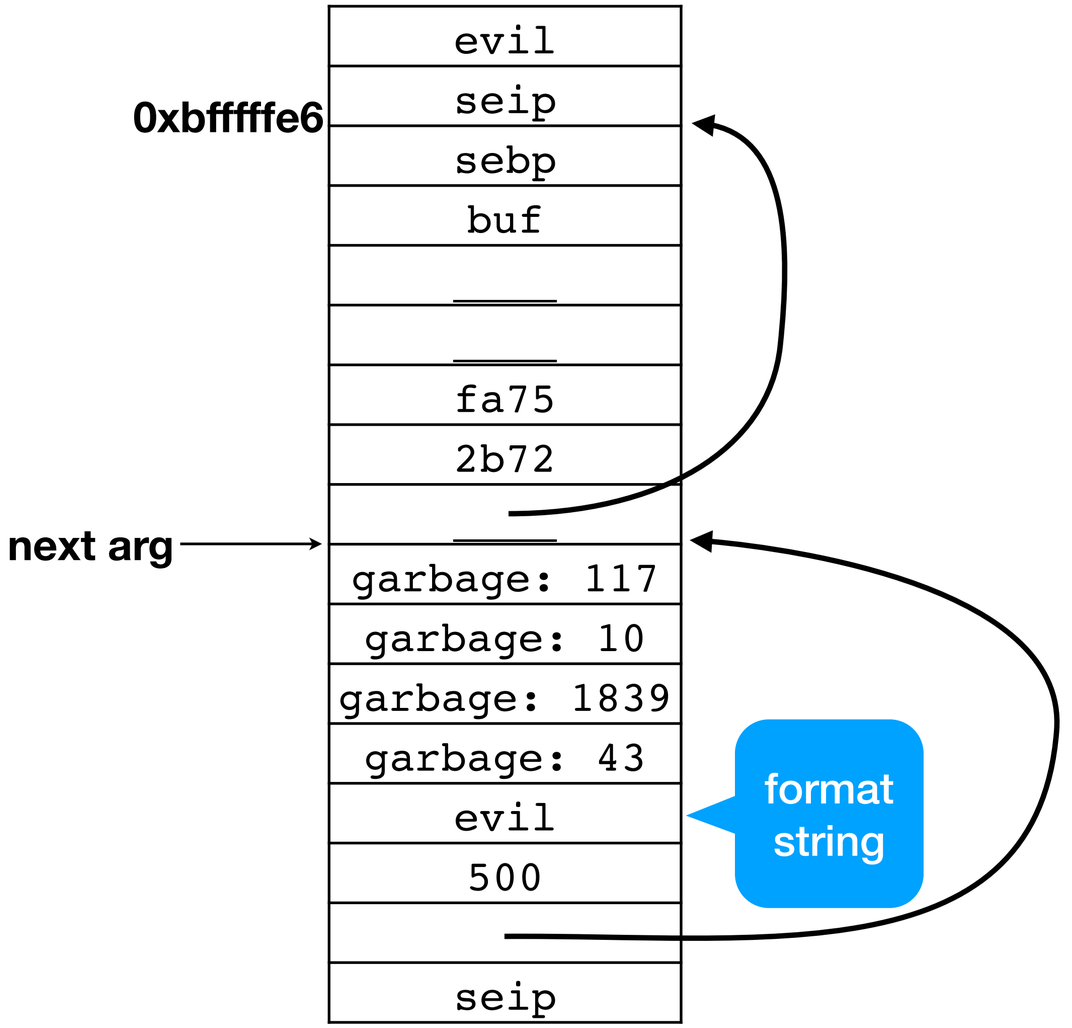
Overwriting seip

```
void foo(const char *evil) {
  char buf[500];
  snprintf(buf, 500, evil);
}
…
foo(“\xe6\xff\xff\xbf%x%x%x%x%n”);
```



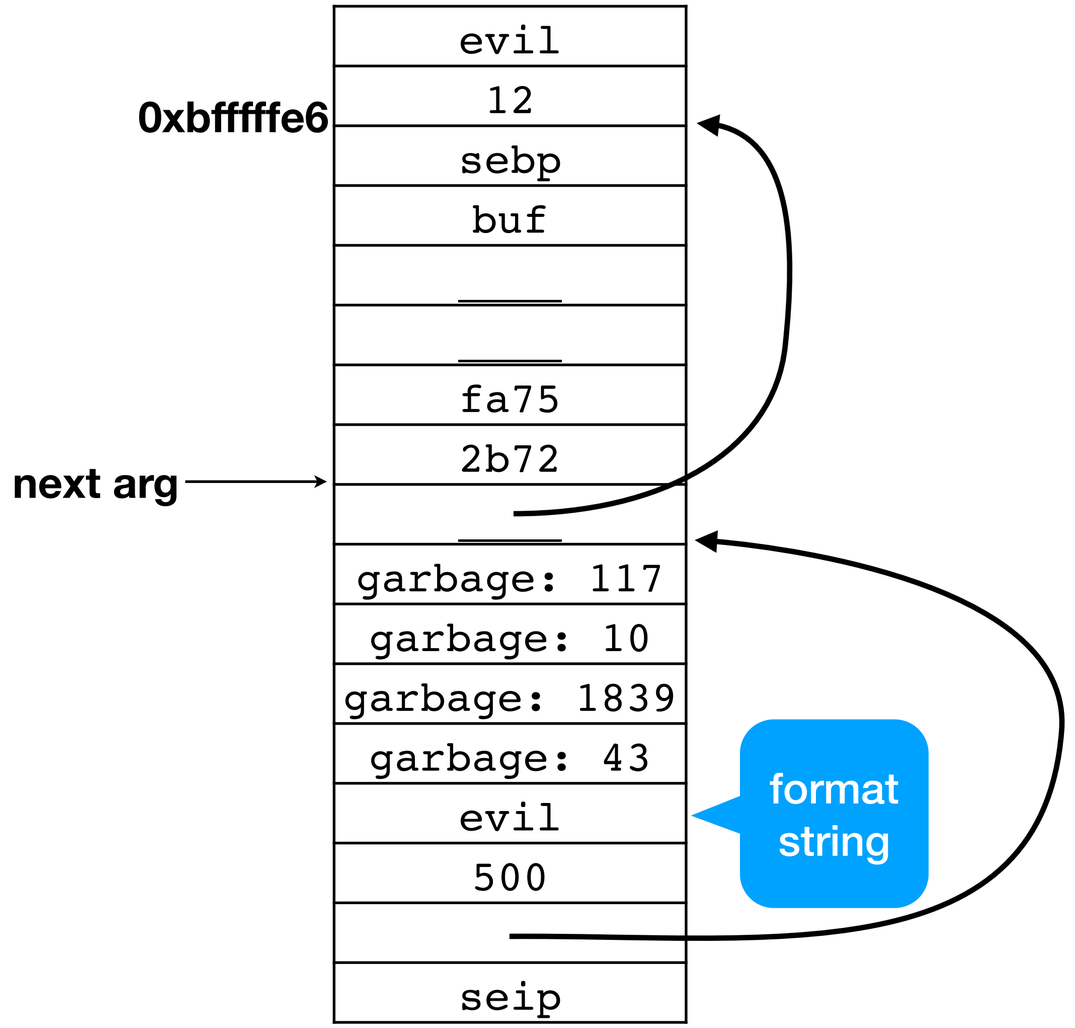
Overwriting seip

```
void foo(const char *evil) {
  char buf[500];
  snprintf(buf, 500, evil);
}
…
foo(“\xe6\xff\xff\xbf%x%x%x%x%n”);
```



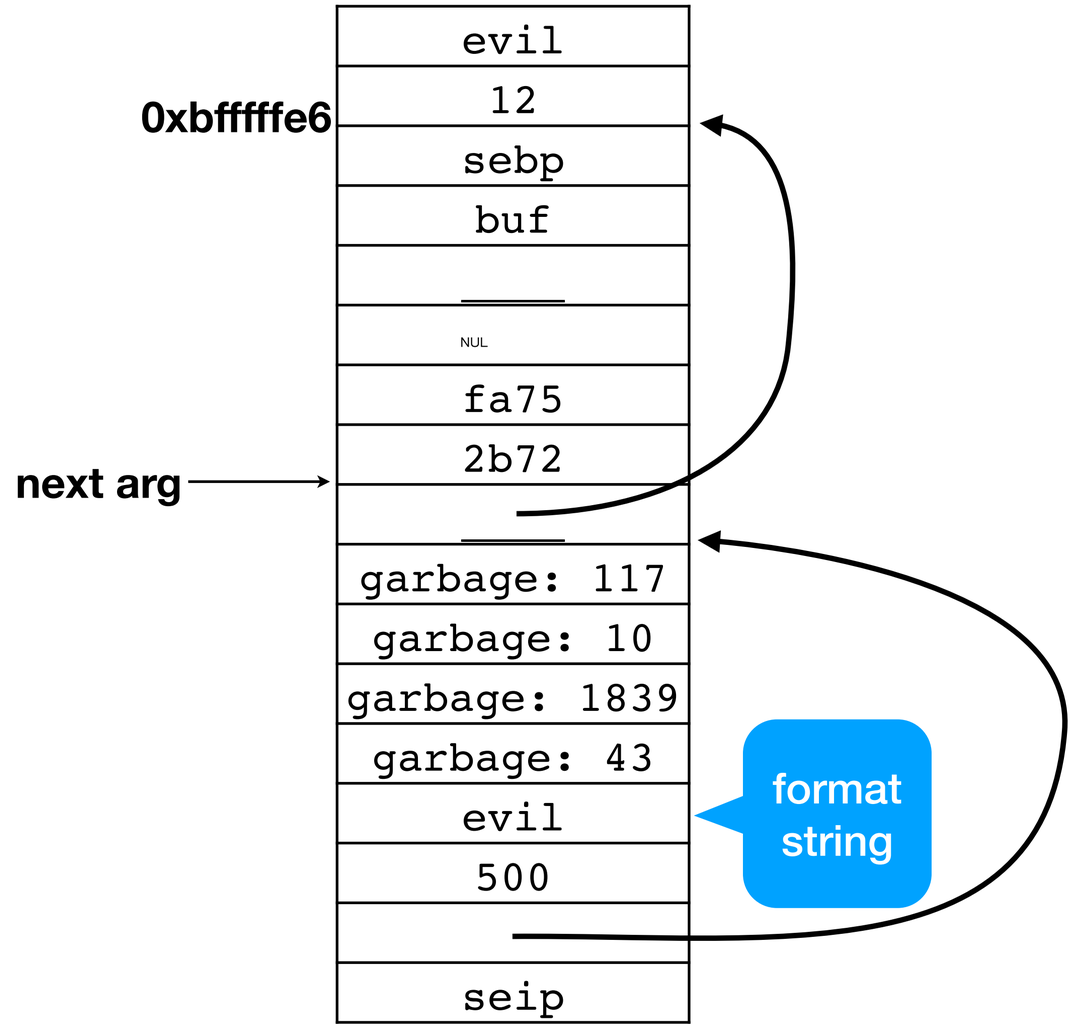
Overwriting seip

```
void foo(const char *evil) {
  char buf[500];
  snprintf(buf, 500, evil);
}
…
foo(“\xe6\xff\xff\xbf%x%x%x%x%n”);
```



Overwriting seip

```
void foo(const char *evil) {
  char buf[500];
  snprintf(buf, 500, evil);
}
…
foo(“\xe6\xff\xff\xbf%x%x%x%x%n”);
```



Picking the bytes to write

- Use `%⟨len⟩x` to control the length of the output
  - E.g., %100x will write the next argument in hexadecimal using 100 characters
  - Note that smaller lengths than are required to print the number are ignored! E.g., %2x for the number 0x12345678 will still print 8 characters
- Use `%hhn` to write just the least-significant byte of the length
  - I.e., it'll write the length modulo 256 to the byte in memory pointed to by the next argument

Almost putting it all together

```
evil = “⟨address⟩ZZZZ”
       “⟨address+1⟩ZZZZ”
       “⟨address+2⟩ZZZZ”
       “⟨address+3⟩”
       “%8x%8x…%8x”
       “%⟨len⟩x%hhn”
       “%⟨len⟩x%hhn”
       “%⟨len⟩x%hhn”
       “%⟨len⟩x%hhn”;
```

By carefully picking the len values, one can write an arbitrary 4 bytes to an arbitrary address

Each address (which cannot contain 0 bytes!) is written to the buffer in little endian, separated by 4 arbitrary characters (I selected *Z*)

The `%8x` move the next argument pointer up the stack until it reaches 4 bytes below the buffer, writing 8 bytes each time

`%⟨len⟩x` will write the next argument in len bytes

`%hhn` will use the next argument, `⟨address⟩`, to store the length so far modulo 256 at address, this repeats for address+1, address+2, and address+3

Misaligned buf

- If `buf` is not 4-byte aligned, prepend 1, 2, or 3 characters to `evil`

x86 is great and all, but what about x86-64?

Bad news: Unlike 32-bit addresses, every x86-64 address you encounter contains 0 bytes

E.g., Here are some address ranges from the cat program

- stack addresses: `0x00007fffffffde000–0x00007ffffffff000`
- code addresses: `0x000055555555e000–0x000055555555b000`
- heap addresses: `0x000055555556e000–0x000055555558e000`

We can’t write addresses and then use them ☹️

Sometimes we can make use of addresses already on the stack

Sometimes we can arrange for addresses to appear on the stack

- Only string operations like strcpy()/strcat() are the ones that cannot have 0 bytes
- Others, like fgets(), and getline() don’t care about 0 bytes and only care about newline bytes (0x0A)
- Files read from disk, either using read() or fread() happily work with binary data and so can contain 0 bytes

One exploitation option

If the target program reads from an attacker-controlled file into a stack buffer, the attacker can put addresses in the file and they'll appear on the stack

An attacker-controlled format string (which could be from the same file or something else entirely) can use those addresses along with %hhn to write bytes to memory at those addresses

More complicated exploits are possible making use of existing addresses on the stack without introducing new ones

Further simplifications

Conversion specifiers (like `%d`, `%16lx`, or `%hhn`) can reference arguments out of order with the syntax `%⟨arg_num⟩$d`

E.g., `%35$16lx` means to take argument 35 as a long (so 8 bytes on x86-64) and print it in 16 hexadecimal characters

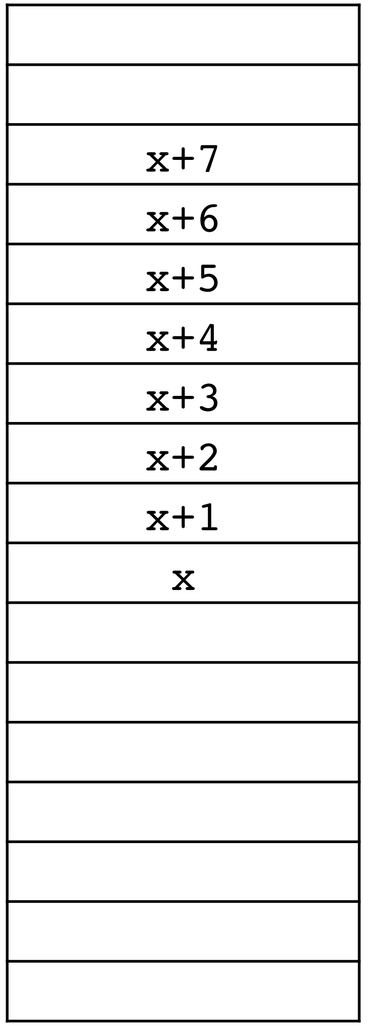
We can use that to selectively print out words on the stack (this is *really* worth remembering)

%10$231c will take the 10th argument and print it as a character in 231 bytes

This can often be easier to use than doing a bunch of %16lx to move the next argument pointer up the stack while writing 16 bytes each time

x86-64 format string exploitation (one approach)

- Want to write the 8 bytes 0x0123456789ABCDEF at address x
- Arrange for the addresses x, x+1, x+2, x+3, x+4, x+5, x+6, and x+7 to appear on the stack **somehow** (we can't use the traditional approach of putting them in the format string because of the 0 bytes in addresses)
- Use `%1$239c` in the format string to write 239 = 0xEF characters of output followed by `%25$hhn` to store 0xEF at address x
- Use `%1$222c%26$hhn` to write 222 more characters and then store (239 + 222) % 256 = 0xCD at address x+1
- Repeat for the other 6 bytes.



Suggestions

Don’t compute the values 239 and 222 by hand

- In your exploit.py, work out the lengths for the `%1$⟨len⟩c` programmatically
- If you have to change the value you want to write, you don’t have to recompute the
- Don’t forget that if your format string starts with anything other than the `%1$239c%25$hhn%1$222c%26$hhn`…
  you’ll need to take the length of the stuff before into account

First figure out what argument number you need (the 25 in %25$hhn) by using format strings like `%10$16lx–%11$16lx–%12$16lx–%13$16lx`…
until you see the addresses you wrote to the stack prior to the format string exploit

Advantages of format string exploits

No need to smash the stack (targeted write)

Avoids defenses such as stack canaries!

- Stack canary is a random word pushed onto the stack that is checked before the function returns

They can be used to read arbitrary locations on the stack (again, remember this!)

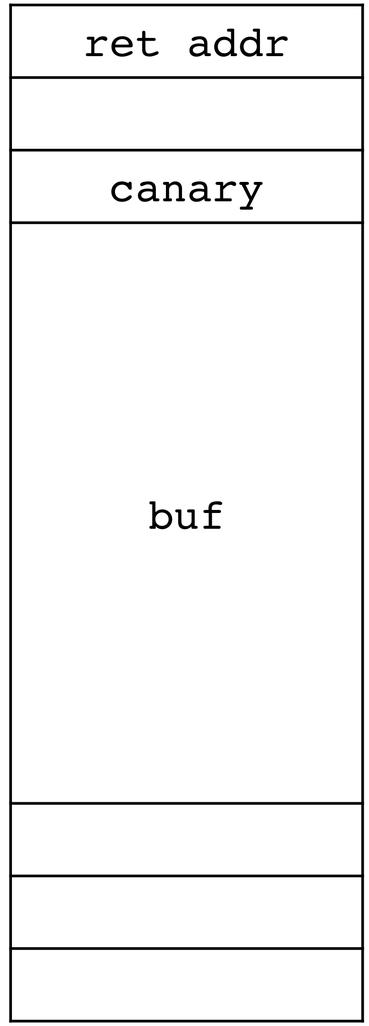
Stack Canaries

```
int bar(char *);
char foo(void) {
    char buf[100];
    bar(buf);
    return buf[0];
}
```

```
foo:
        sub     rsp, 120
        mov     rax, QWORD PTR fs:40
        mov     QWORD PTR [rsp+104], rax
        xor     eax, eax
        mov     rdi, rsp
        call    bar
        movzx   eax, BYTE PTR [rsp]
        mov     rdx, QWORD PTR [rsp+104]
        sub     rdx, QWORD PTR fs:40
        jne     .L5
        add     rsp, 120
        ret
.L5:
        call    __stack_chk_fail
```

fs is a segment register which, on x86-64 is essentially just a (complicated) pointer

fs:40 means take the value 40 bytes from the start of the fs segment



Disadvantages of format string exploits

Format string vulnerabilities are *very* easy for the compiler to spot and warn about which makes them comparatively rare relative to other vulnerabilities

- ```
  $ gcc -Wformat=2 f.c
  f.c: In function ‘main’:
  f.c:5: warning: format not a string literal and no
  format arguments
  ```

x86-64 makes format string exploits tricky to write, especially if you don’t have the ability to put arbitrary data containing 0s on the stack
  ```

What else can we overwrite?

- Function pointers
- C++ vtables
- Global offset table (GOT)

Function pointers

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

int compare(const void *a,
            const void *b) {
  const int *x = a;
  const int *y = b;
  return *x - *y;
}

int main() {
  int i;
  int arr[6] = {2, 1, 5, 13, 8, 4};
  qsort(arr, 6, 4, compare);
  for (i = 0; i < 6; ++i)
    printf("%d ", arr[i]);
  putchar('\n');
  return 0;
}
```

```
main:
        …
        mov     rbx, rsp
        lea     rcx, compare[rip]
        mov     edx, 4
        mov     esi, 6
        mov     rdi, rbx
        call    qsort@PLT
        …
qsort:
        …
        call rax
        …
```

C++ Virtual function tables (vtable)

```
struct Foo {
  Foo() { }
  virtual ~Foo() { }
  virtual void fun1() { }
  virtual void fun2() { }
};

void bar(Foo &f) {
  f.fun1();
  f.fun2();
}

int main() {
  Foo f;
  bar(f);
}
```

```
_Z3barR3Foo:
        push    rbx
        mov     rbx, rdi                // rbx = this
        mov     rax, QWORD PTR [rdi]    // rax = this->vptr
        call    QWORD PTR [rax+16]      // call Foo::fun1(this)
        mov     rax, QWORD PTR [rbx]    // rax = this->vptr
        mov     rdi, rbx                // rdi = this
        call    QWORD PTR [rax+24]      // call Foo::fun2(this)
        pop     rbx
        ret
```

vptr points to vtable + 16

```
_ZTV3Foo:
        .quad   0
        .quad   _ZTI3Foo
        .quad   _ZN3FooD1Ev
        .quad   _ZN3FooD0Ev
        .quad   _ZN3Foo4fun1Ev
        .quad   _ZN3Foo4fun2Ev

// Demangled
vtable for Foo:
        .quad 0
        .quad typeinfo for Foo
        .quad Foo::~Foo()
        .quad Foo::~Foo()
        .quad Foo::fun1()
        .quad Foo::fun2()
```

```
main:
        sub     rsp, 24
        lea     rax, vtable for Foo[rip+16]
        mov     QWORD PTR [rsp+8], rax
        lea     rdi, [rsp+8]
        call    _Z3barR3Foo
        mov     eax, 0
        add     rsp, 24
        ret
```

Stores the address of vtable + 16 in the vptr of the Foo object (which here is at rsp + 8)

Global Offset Table (GOT)

- Contains pointers to code and data in shared libraries
- Library functions aren’t called directly; stub in the Procedure Linkage Table (PLT) called
- E.g., call exit -> call exit@plt
- exit@plt looks up the address of exit in the GOT and jumps to it (not the whole story)
- Overwrite function pointer in GOT

Exploitation techniques and Project 1

Exploitation techniques we've seen

Buffer overflows

- Overwrite the saved return address on the stack with the address of shellcode [demo]
- Overwrite *something else* on the stack like one of those other code pointers we talked about
- Overwrite *something else* like data that is used to make control-flow decisions

Integer overflow

- Use to bypass length checks; combine with something else, like a buffer overflow

Format string exploits

Project 1 hints

At this point, you know enough to solve targets 1, 2, 3, 4, and 6

The comments in the targets' source code about safety or correctness may be aspirational rather than factual; in other words: **don't trust that they're correct**

Real programs often have error handling code that is not as robustly checked as the "happy path" through the code; sometimes **inducing errors to take the error handling path is the way to the vulnerability**

Some project 1 specifics

Target 1 is a buffer overflow on the stack

Targets 2 and 3 require at least one other exploitation technique

Target 4 has stack canaries so if you overflow a buffer, for example, the program will abort when the function containing the overflown buffer returns, but not earlier

Target 5 will require exploiting the custom malloc implementation in smalloc.c; we'll talk about this technique (which is more generally applicable than just memory allocators on Monday)

Target 6 is a sudo-like program; you just need to convince the program you're authorized; don't bother trying to crack the password hash, that is unlikely to be successful (although I admit, I haven't tried it myself)