CSE127, Computer Security

Control flow vulnerabilities continued: format strings, integer overflow

UC San Diego

Housekeeping

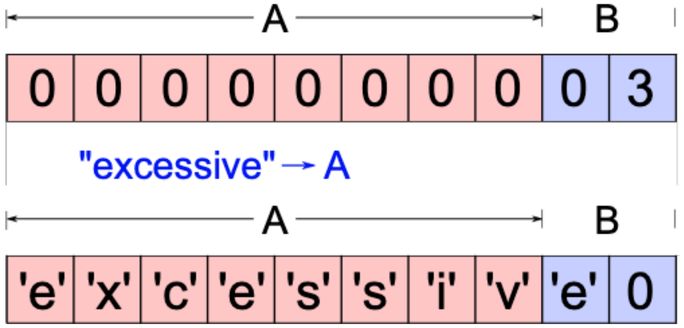
General course things to know

- PA1 due **TONIGHT** by **1/15** at 11:59
  - Good luck!
  - ***Do not forget to submit your AI attestation!***
- Due **1/16** at 11:59
  - #FinAid Canvas quiz: https://canvas.ucsd.edu/courses/71475/quizzes/238979, reminder to do this!
- PA2 releases Friday at midnight, due **1/27** at 11:59

Previously on CSE127…

Buffer overflows!

- We talked about software exploits, control flow, and buffer overflows on stack…
  - Mixing of code / data in the runtime allows for all kinds of “weird” behavior, crashing programs, at worst, **overwriting return address….**



Addressing buffer overflows

- Best way to avoid these bugs is to not have them in the first place
  - If you can, avoid C/C++ for systems programming. Use a memory-safe language instead (Rust, Go)
  - Train developers to understand these bugs and their ramifications (can only really get you so far)
- Finding bugs is a must
  - Manual code review, static analysis, fuzzing, etc., — this is a **whole subfield of computer security**
- Or… we can make the bugs harder to exploit
  - More on this in one lecture (next time: AppSec defenses)

Today's lecture – More software vulns

Learning Objectives

- Understand variadic functions in C
- Learn how format strings really work, and how you can exploit format strings (and printf) to **read and write** arbitrarily on the stack
- Understand the basic concepts of integer overflows, what we can do with them, and why they matter

Format String Vulnerabilities

`printf`

- What does the following code do?

```
printf(“Hello world!”);
```

printf

- What does the following code do?

```
printf(“Hello world!”);
```

- What about this?

```
printf(“%s”, “Hello world!”);
```

printf

- What does the following code do?

```
printf(“Hello world!”);
```

- What about this?

```
printf(“%s”, “Hello world!”);
```

- What about this?

```
printf(“Hello world!%s”);
```

`printf` API

- `printf(“Diagnostic number: %d, message: %s\n”, j, buf)`
  - *“If format includes **format specifiers** (subsequences beginning with %), the additional arguments following format are formatted and inserted in the resulting string replacing their respective specifiers.”*
- A format specifier follows this prototype:
  - `%[flags][width][.precision][length]specifier`

`printf` API

`%[flags][width][.precision][length]specifier`

- Flags
  - **-** Left-justify within the given field width
  - **+** Precede the result with a plus or minus sign
  - **0** Left-pads the number with zeroes (0) instead of spaces when padding is specified

`printf` API

`%[flags]`**`[width]`**`[.precision][length]specifier`

- Width
  - <Number> — minimum characters to be printed. If value is shorter than number, print blank spaces

`printf` API

`%[flags]`**`[width]`**`[.precision][length]specifier`

- Width
  - <Number> — minimum characters to be printed. If value is shorter than number, print blank spaces
- What will the following print?

```
printf("%4s", "H");
```

`printf` API

`%[flags]`**`[width]`**`[.precision][length]specifier`

- Width
  - <Number> — minimum characters to be printed. If value is shorter than number, print blank spaces
- What will the following print?

```
printf("%4s", "H");
```

"<space><space><space>H"

`printf` API

`%[flags][width]`**`[.precision]`**`[length]specifier`

- Precision
  - .<number> — for integer specifiers, precision specifies the minimum number of digits to be written; if value is shorter than number, pad with zeroes
  - For s, maximum number of characters to be printed, **default is print until ending null character is encountered**
- What will the following print?

```
printf("%.3s", "Hello");
```

`printf` API

`%[flags][width][.precision]`**`[length]`**`specifier`

- Length
  - *Modifies the length of the data type provided* (doing conversions if necessary… more on this later)
    - `h: char`
    - `hh: short`
    - `l: long`
    - `ll: long long`

`printf` API continued

An introduction to variadic functions

```
int printf(const char *format, ...)
```

- How many arguments does `printf` take?

`printf` API continued

An introduction to variadic functions

```
int printf(const char *format, ...)
```

- How many arguments does `printf` take?
  - It depends on you!
- C supports functions with a ***variable number*** of arguments, called *variadic functions*

Variadic functions

How do we implement them?

- Question: If the number of arguments is not pre-determined for variadic functions, then **how does the called function know how many were passed in?**

Variadic functions

How do we implement them?

- Question: If the number of arguments is not pre-determined for variadic functions, then **how does the called function know how many were passed in?**

- `printf` works by *parsing the format string during runtime, and keeping track of necessary stack variables independently*

va_list

Defined in header <stdarg.h>

```
/* unspecified */ va_list;
```

va_list is a complete object type suitable for holding the information needed by the macros va_start, va_copy, va_arg, and va_end.

If a va_list instance is created, passed to another function, and used via va_arg in that function, then any subsequent use in the calling function should be preceded by a call to va_end.

It is legal to pass a pointer to a va_list object to another function and then use that object after the function returns.

Simple printf implementation

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

    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;
            }
        }
    }
}
```

Simple printf implementation

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

    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;
            }
        }
    }
}
```

printf keeps an “argument pointer,” sort of like an internal stack pointer…

Simple printf implementation

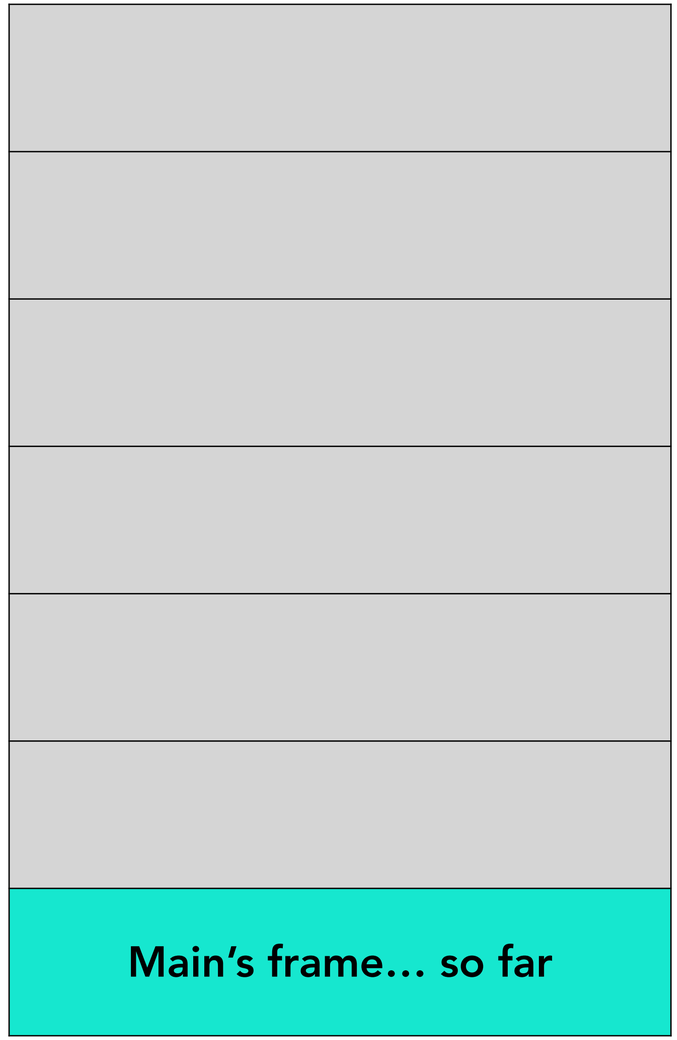
```
int printf(const char* format, ...) {
    int i; char c; char *s; double d;
    va_list ap; /* Declare an "argument pointer" to a variable argument list */
    va_start(ap, format); /* Iniitalize argument pointer using last known argument*/

    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;
            }
        }
    }
}
```

Function parses out arguments based on the format string itself

`printf` on the stack

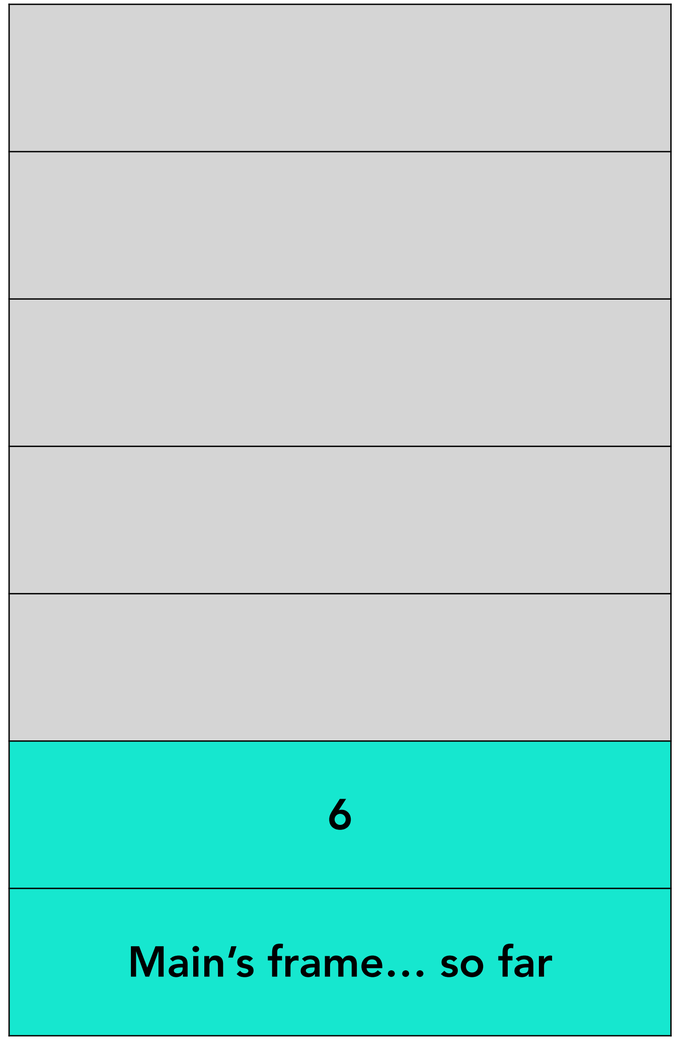
```
int main(void) {
    ...
    printf("Numbers: %d,%d", 5, 6);
    return 0;
}
```



main

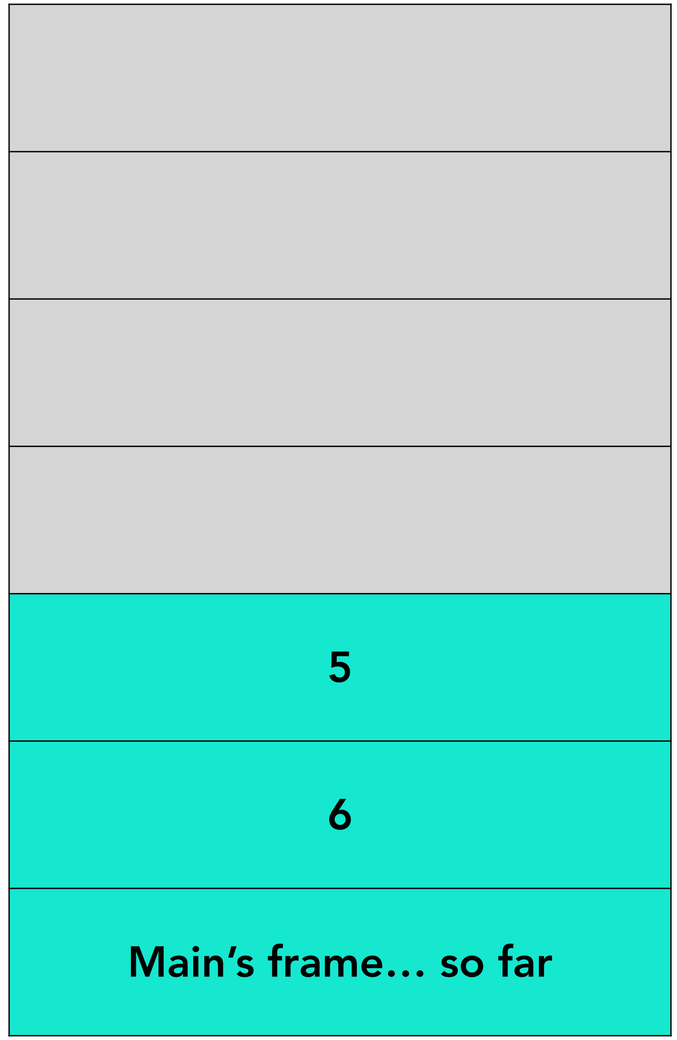
`printf` on the stack

```
int main(void) {
    ...
    printf("Numbers: %d,%d", 5, 6);
    return 0;
}
```



`printf` on the stack

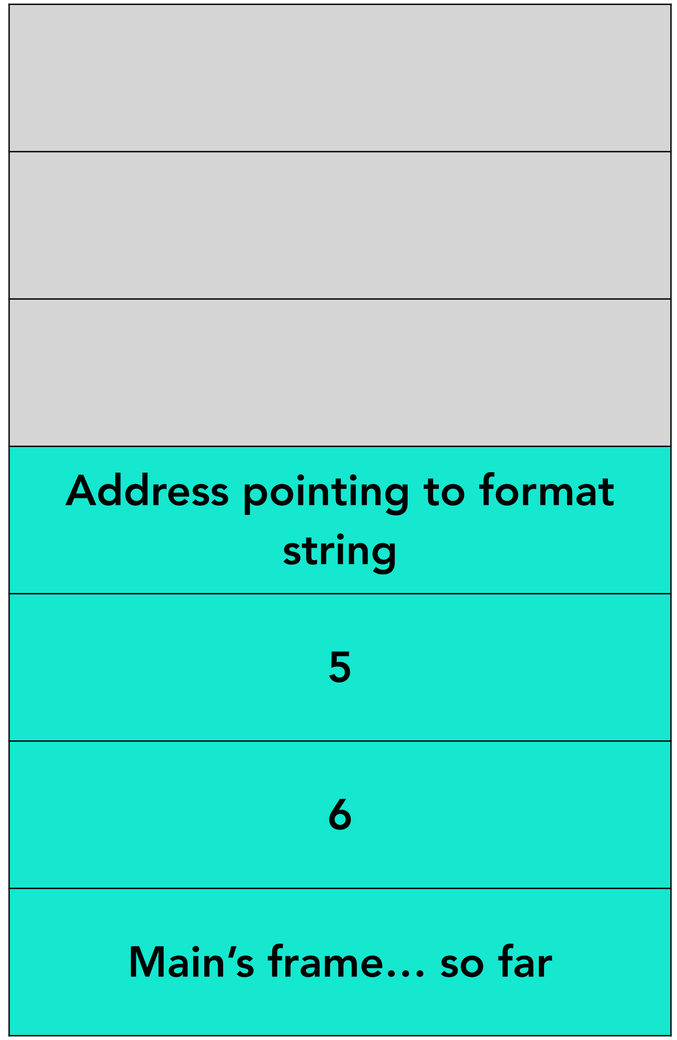
```
int main(void) {
    ...
    printf("Numbers: %d,%d", 5, 6);
    return 0;
}
```



main

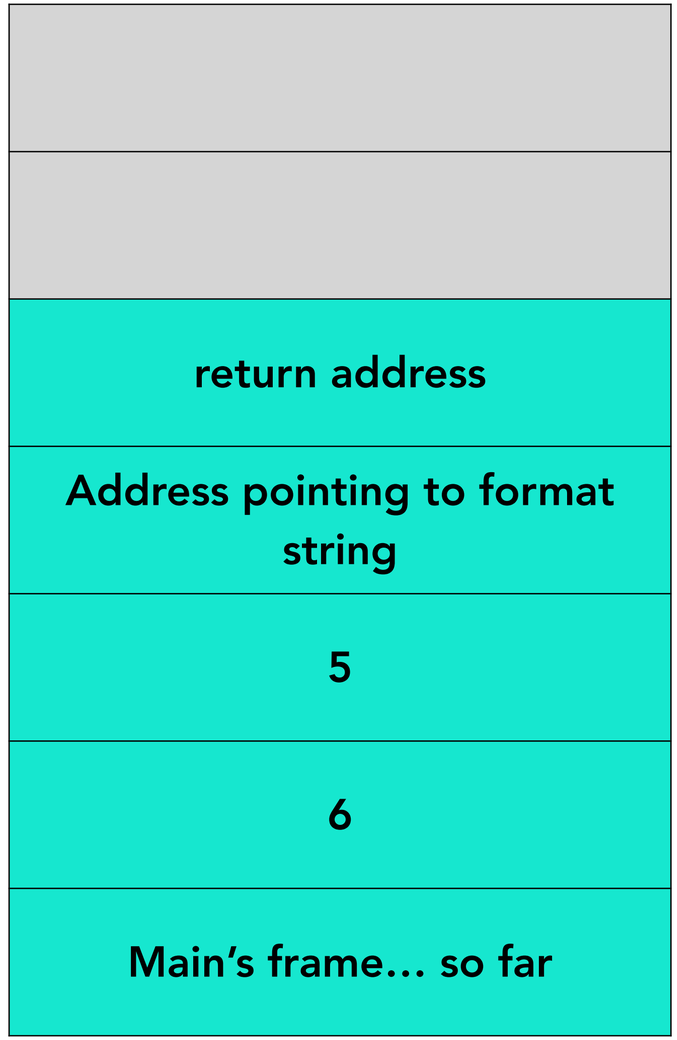
`printf` on the stack

```
int main(void) {
    ...
    printf("Numbers: %d,%d", 5, 6);
    return 0;
}
```



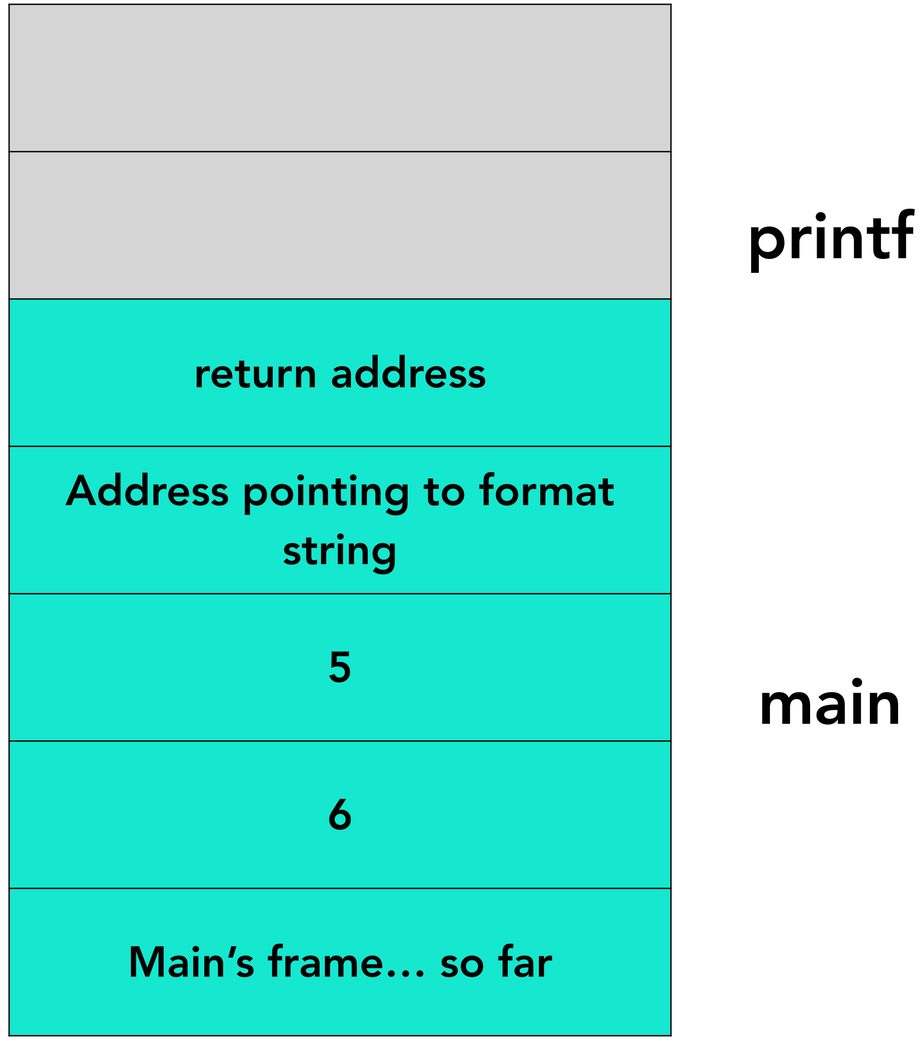
`printf` on the stack

```
int main(void) {
    ...
    printf("Numbers: %d,%d", 5, 6);
    return 0;
}
```



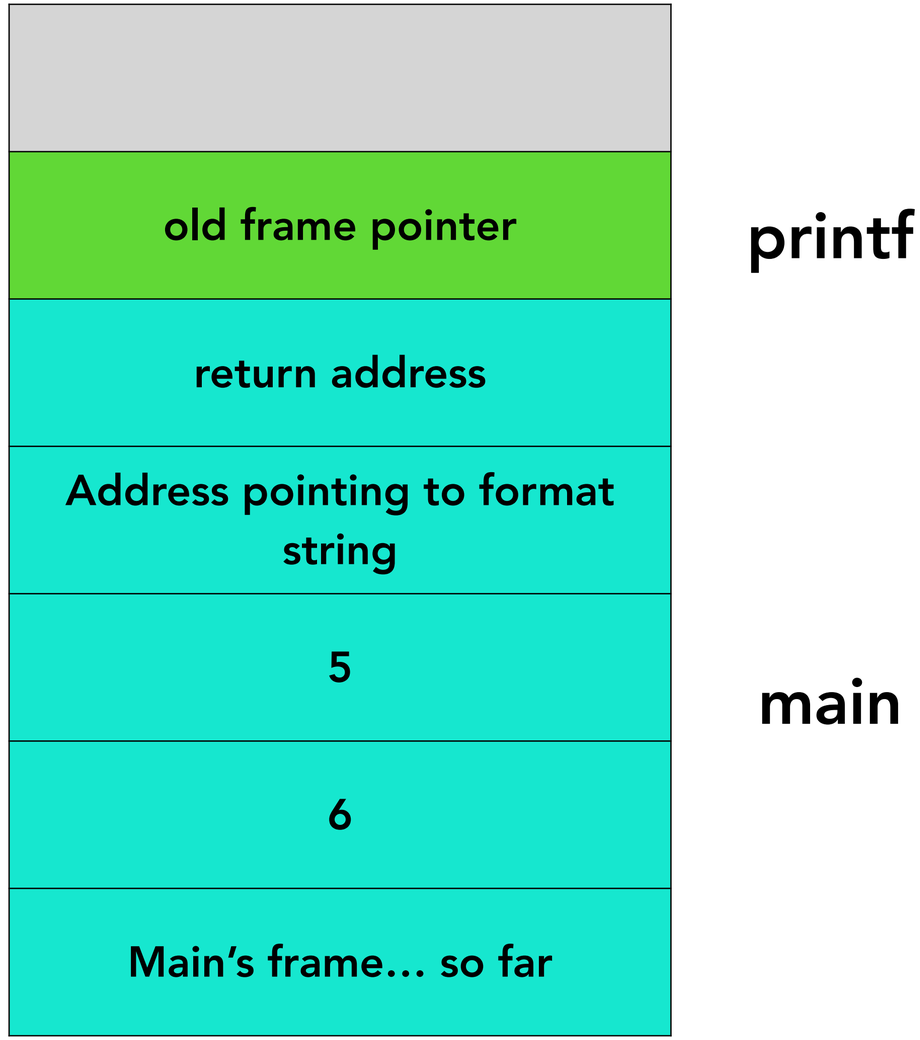
`printf` on the stack

```
int main(void) {
    ...
    printf("Numbers: %d,%d", 5, 6);
    return 0;
}
```



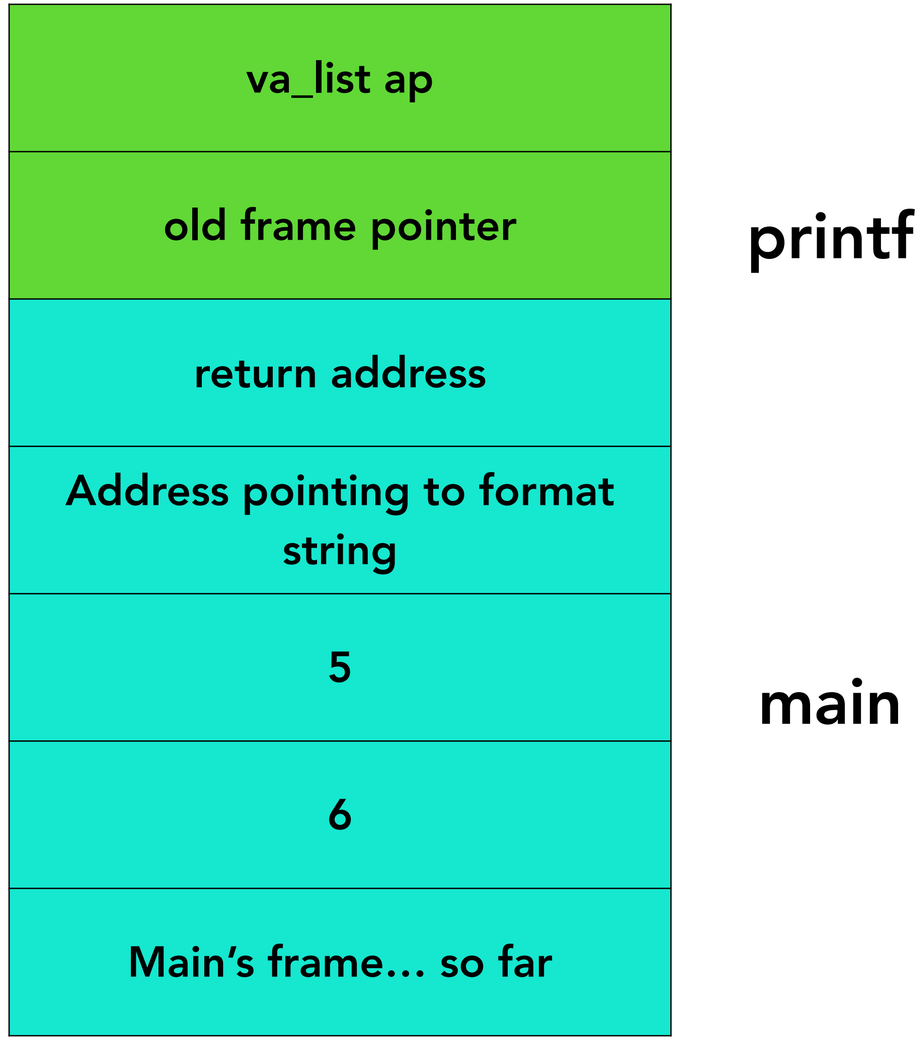
`printf` on the stack

```
int main(void) {
    ...
    printf("Numbers: %d,%d", 5, 6);
    return 0;
}
```



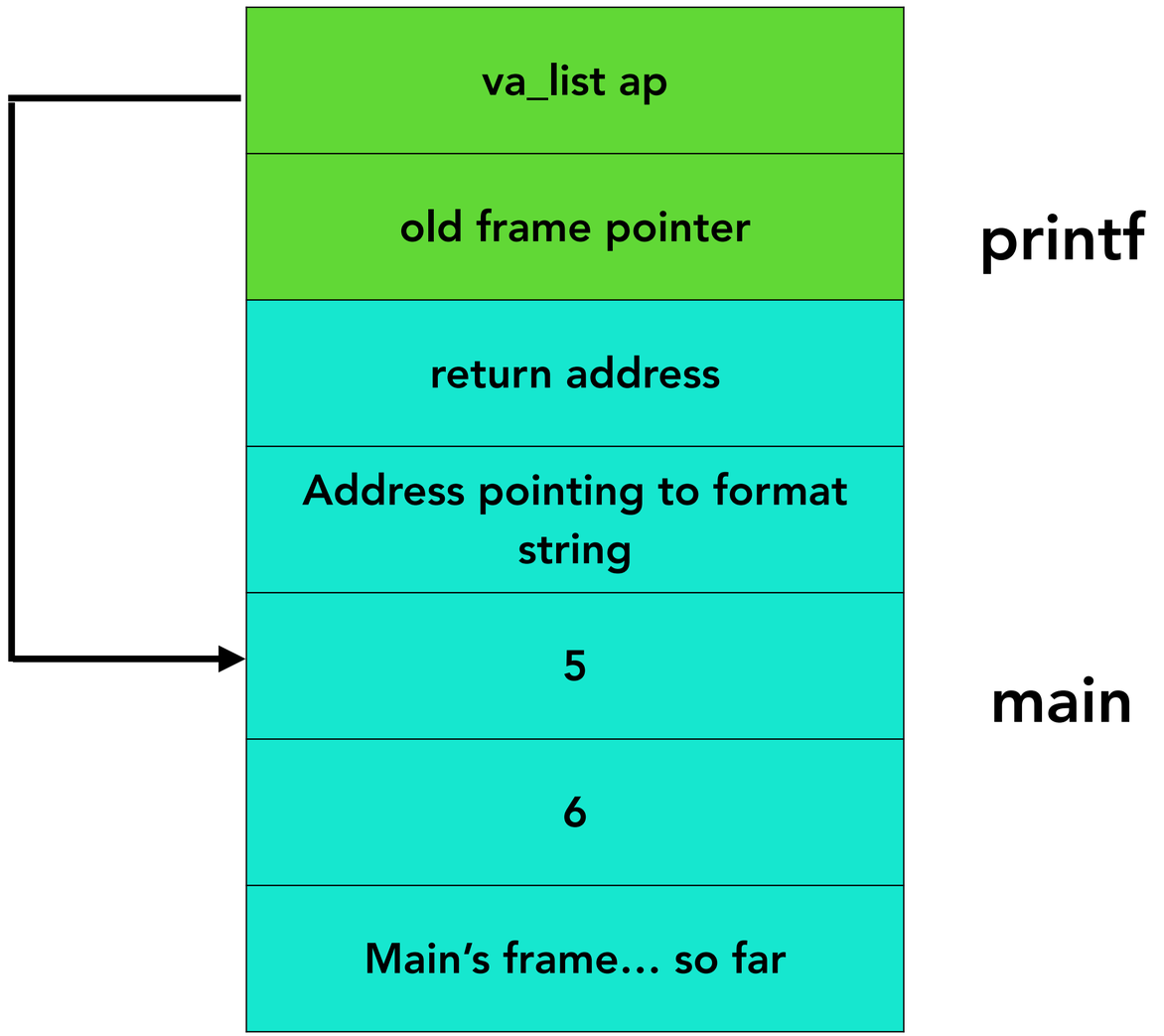
`printf` on the stack

```
int main(void) {
    ...
    printf("Numbers: %d,%d", 5, 6);
    return 0;
}
```



`printf` on the stack

```
int main(void) {
    ...
    printf("Numbers: %d,%d", 5, 6);
    return 0;
}
```

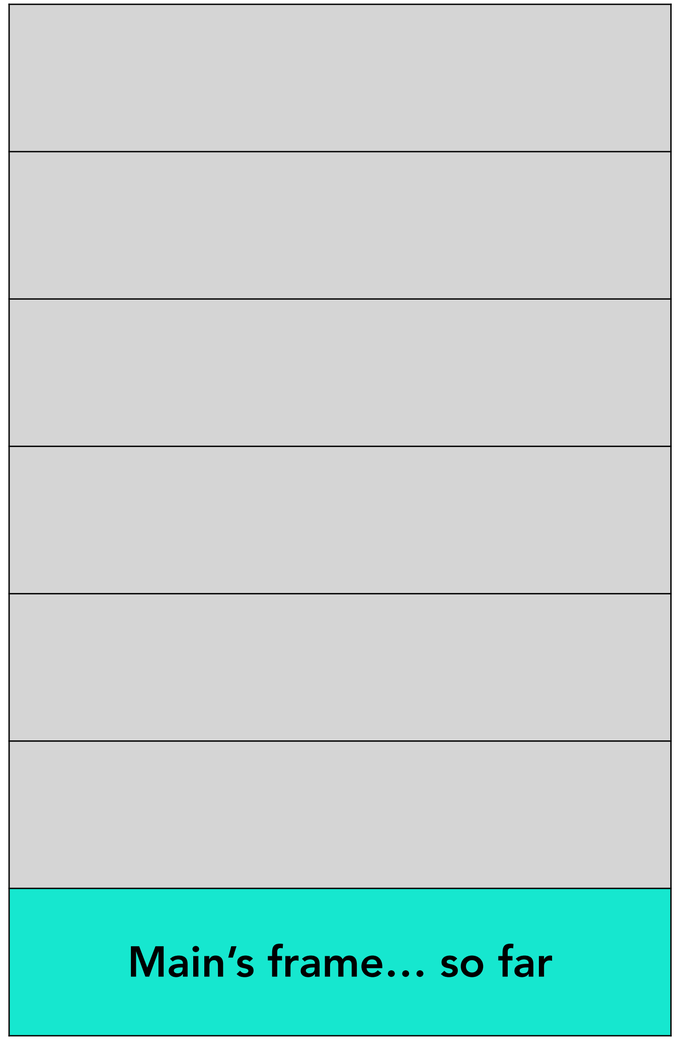


After calling va_start, ap is initialized to *right after the format string* and reads arguments defined by format string

`printf` on the stack, again

This time, I haven't passed in anything to `printf`... what happens?

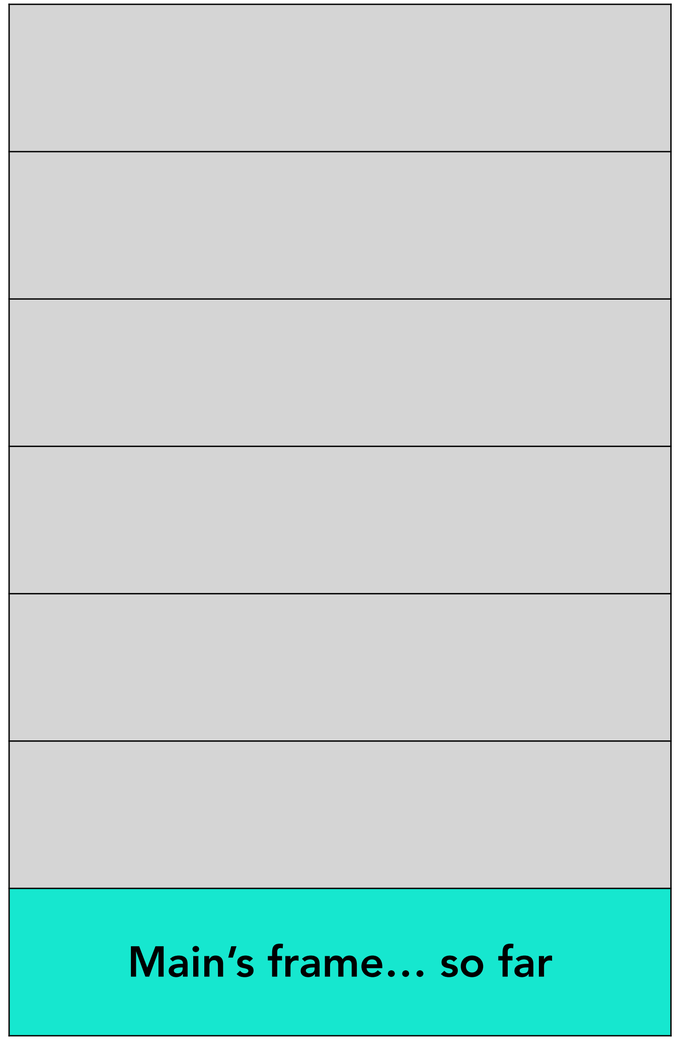
```
int main(void) {
    ...
    printf("Numbers: %d,%d");
    return 0;
}
```



main

`printf` on the stack, again

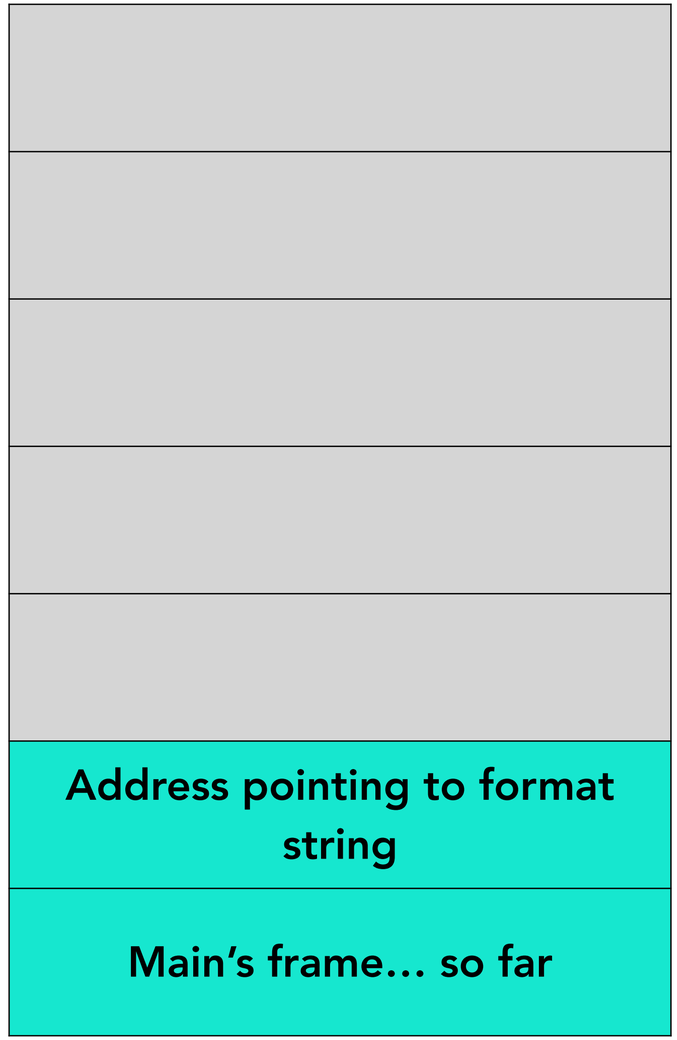
```
int main(void) {
    ...
    printf("Numbers: %d,%d");
    return 0;
}
```



main

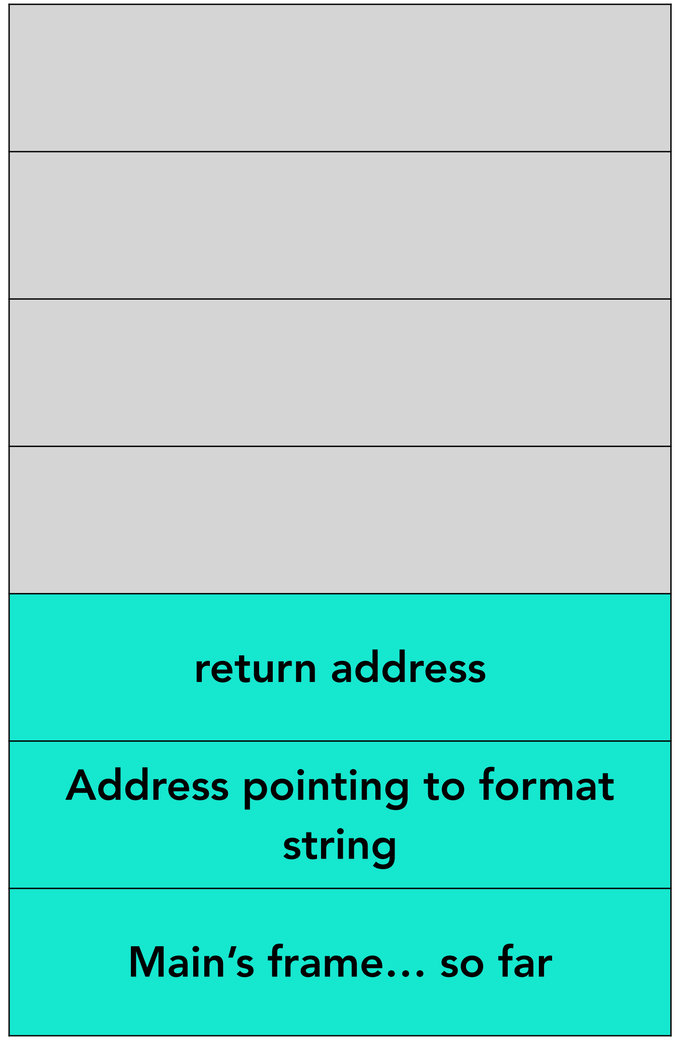
`printf` on the stack, again

```
int main(void) {
    ...
    printf("Numbers: %d,%d");
    return 0;
}
```



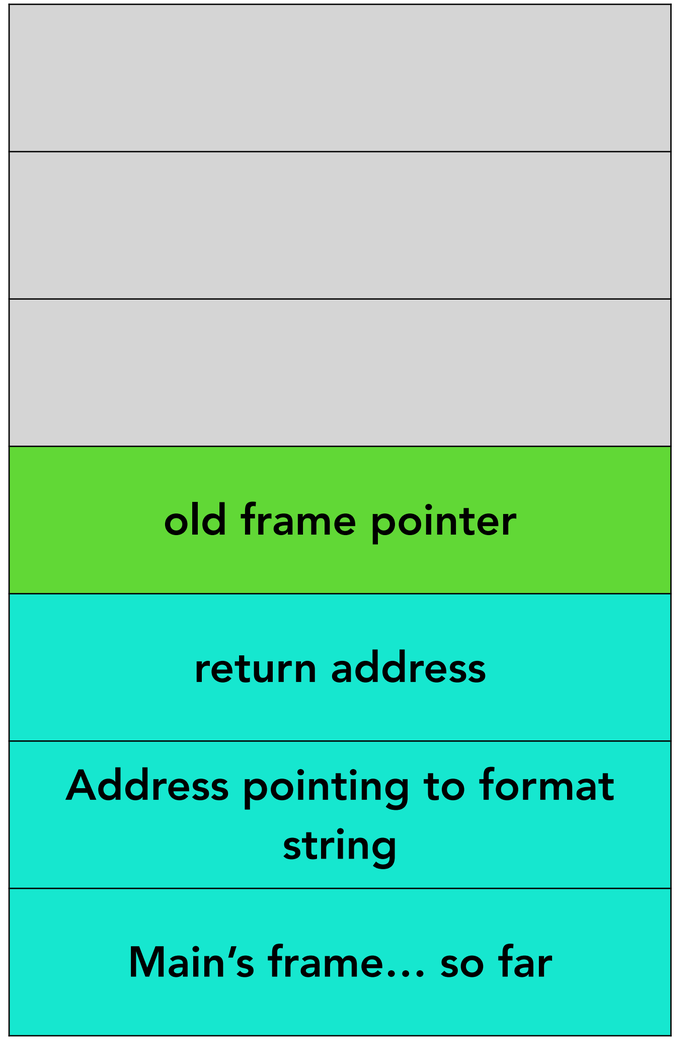
`printf` on the stack, again

```
int main(void) {
    ...
    printf("Numbers: %d,%d");
    return 0;
}
```



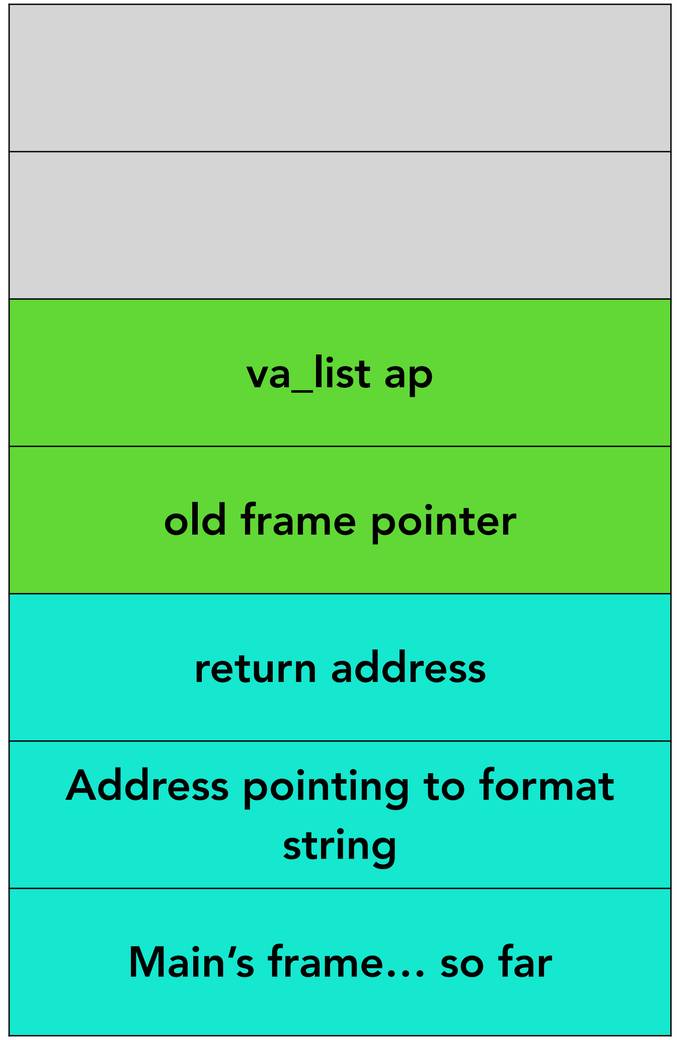
`printf` on the stack, again

```
int main(void) {
    ...
    printf("Numbers: %d,%d");
    return 0;
}
```



`printf` on the stack, again

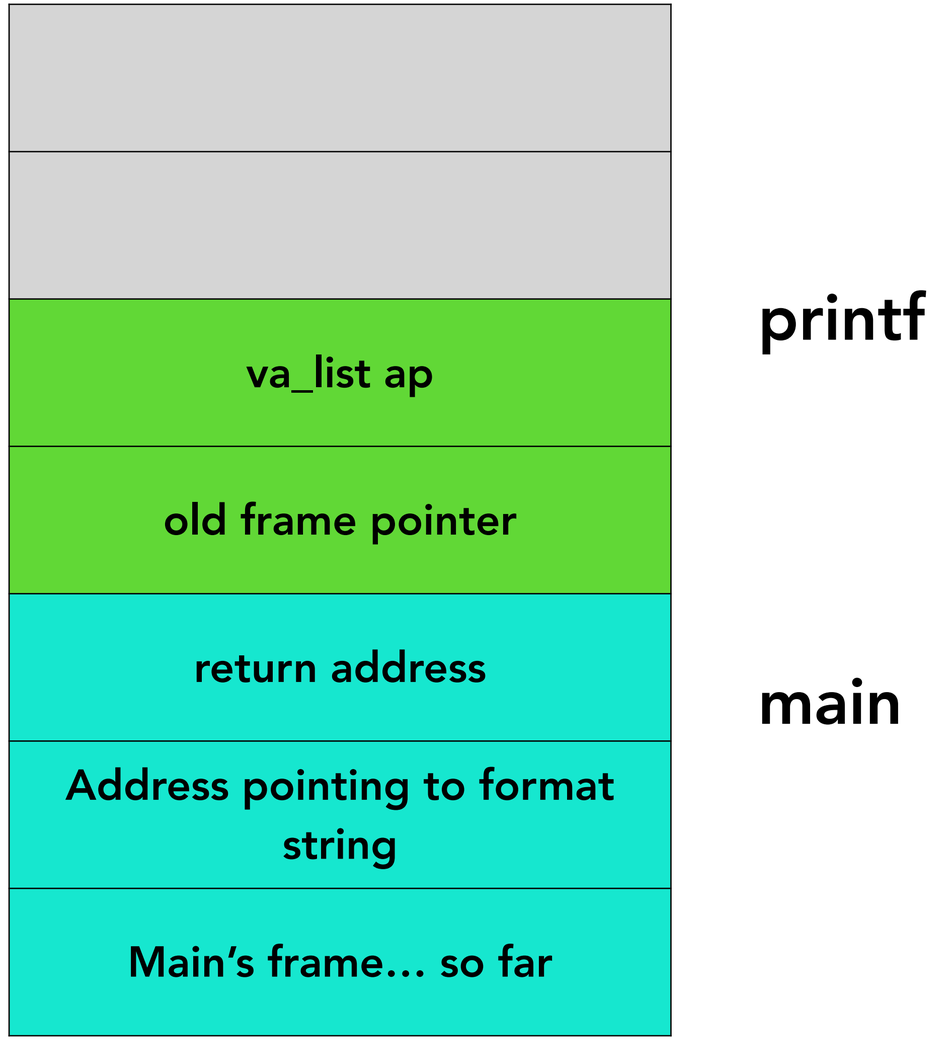
```
int main(void) {
    ...
    printf("Numbers: %d,%d");
    return 0;
}
```



`printf` on the stack, again

```
int main(void) {
    ...
    printf("Numbers: %d,%d");
    return 0;
}
```

Where will va_list ap point to after va_start is called?

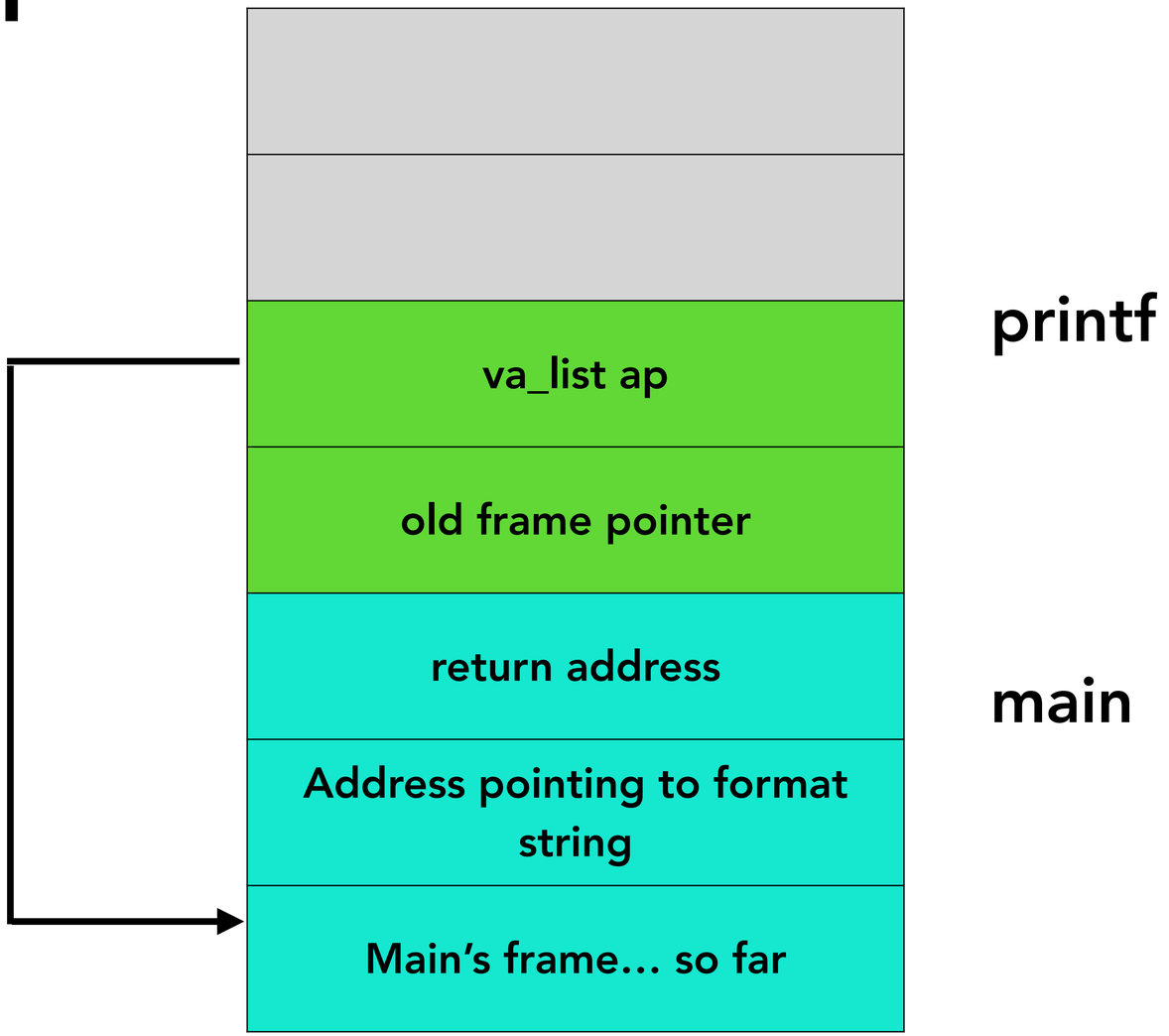


`printf` on the stack, again

```
int main(void) {
    ...
    printf("Numbers: %d,%d");
    return 0;
}
```

Where will va_list ap point to after va_start is called?

Somewhere in main's frame… 😱



Key problems with `printf`

- User is responsible for enforcing one-to-one mapping between format specifier and the arguments passed in
  - Do you trust the user to do this the right way? What about an attacker?
- **`printf`** implements basically its own runtime parser…
  - Parsing is fraught and hard (who wants to parse *anything* these days?)
  - No way to differentiate between *code* and *data*! (variations on a theme….)

Format string vulnerabilities

- Still, how had could it be?
- What can an attacker do with a well-crafted format string?

Format string vulnerabilities

- Still, how had could it be?
- What can an attacker do with a well-crafted format string?
  - Read arbitrary data (bad)

Format string vulnerabilities

- Still, how had could it be?
- What can an attacker do with a well-crafted format string?
  - Read arbitrary data (bad)
  - Write arbitrary data (really bad!!)

Format string vulnerabilities

Reading from the stack

- What does the following do?

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

Format string vulnerabilities

Reading from the stack

- What does the following do?

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

  - Read and print the four words up the stack (above **va_list**)
  - What's up beyond the argument list? Local variables, caller state, etc.…

Format string vulnerabilities

Reading via pointer

- What does the following do?

```
printf("%s\n");
```

Format string vulnerabilities

Reading via pointer

- What does the following do?

```
printf("%s\n");
```

- Take the location va_list points to, interpret it as a char *, and print the memory *at that address* as a string until a null byte is reached.

Format string vulnerabilities

Reading arbitrary memory

```
void f() {
    char localstring[80] =
"\x10\x01\x48\x08_%08x.%08x.|%s|";
    int i, j;
    printf(localstring);
}
```

What does `f` do?

Format string vulnerabilities

Reading arbitrary memory

```
void f() {
    char localstring[80] =
"x10\x01\x48\x08_%08x.%08x.|%s|";
    int i, j;
    printf(localstring);
}
```

First, initializes a character buffer (on stack) with this interesting value…
we'll dissect it later

Format string vulnerabilities

Reading arbitrary memory

```
void f() {
    char localstring[80] =
"x10\x01\x48\x08 %08x.%08x.|%s|";
    int i, j;
    printf(localstring);
}
```

Declares two integers

Format string vulnerabilities

Reading arbitrary memory

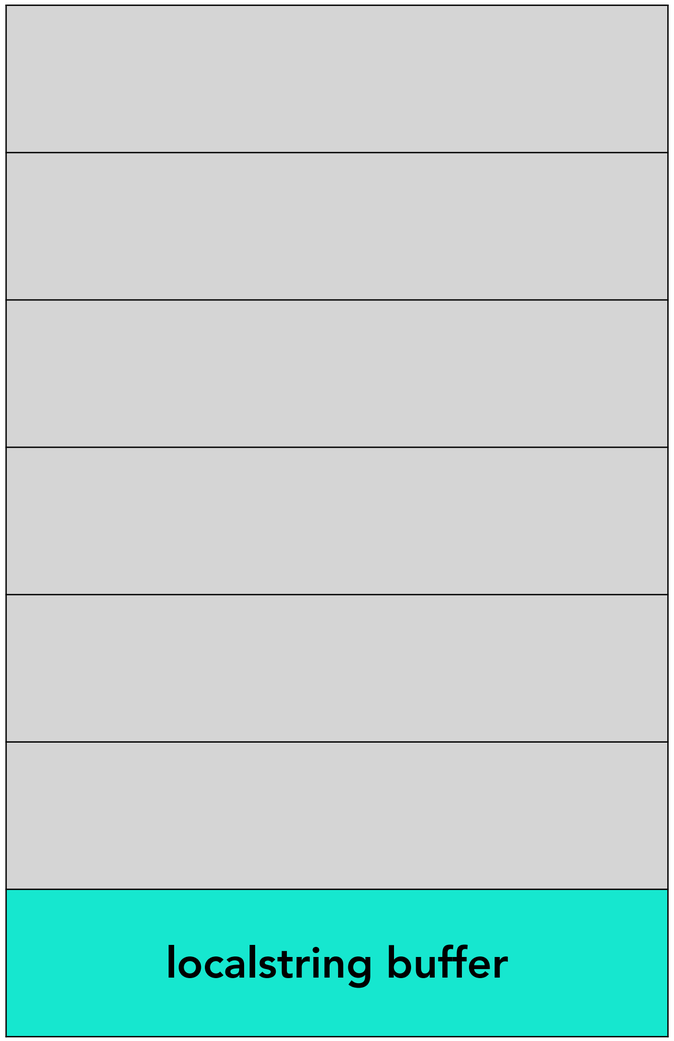
```
void f() {
    char localstring[80] =
"x10\x01\x48\x08_%08x.%08x.|%s|";
    int i, j;
    printf(localstring);
}
```

Calls printf

Format string vulnerabilities

Reading arbitrary memory

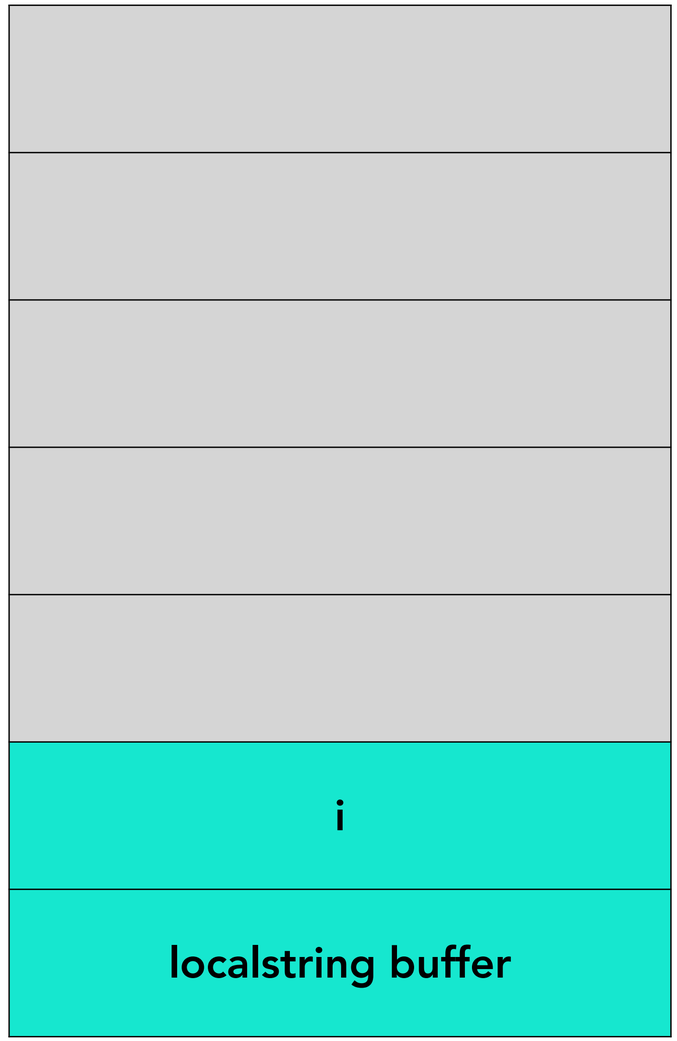
```
void f() {
    char localstring[80] =
"x10\x01\x48\x08_%08x.%08x.|
%s|";
    int i, j;
    printf(localstring);
}
```



Format string vulnerabilities

Reading arbitrary memory

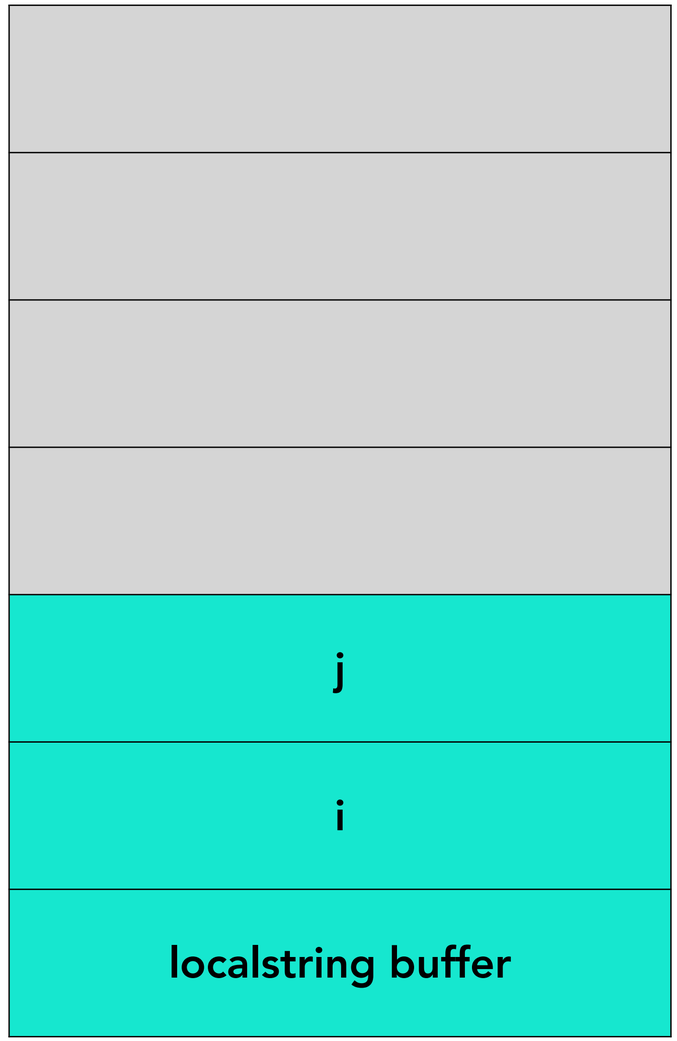
```
void f() {
    char localstring[80] =
"x10\x01\x48\x08_%08x.%08x.|
%s|";
    int i, j;
    printf(localstring);
}
```



Format string vulnerabilities

Reading arbitrary memory

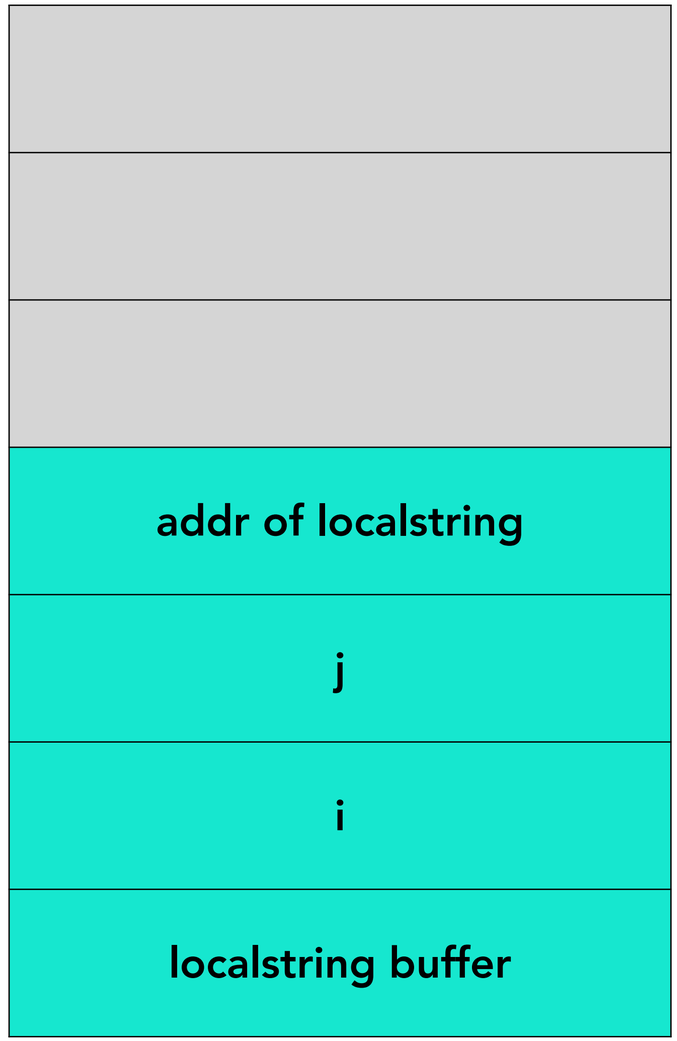
```
void f() {
    char localstring[80] =
"x10\x01\x48\x08_%08x.%08x.|
%s|";
    int i, j;
    printf(localstring);
}
```



Format string vulnerabilities

Reading arbitrary memory

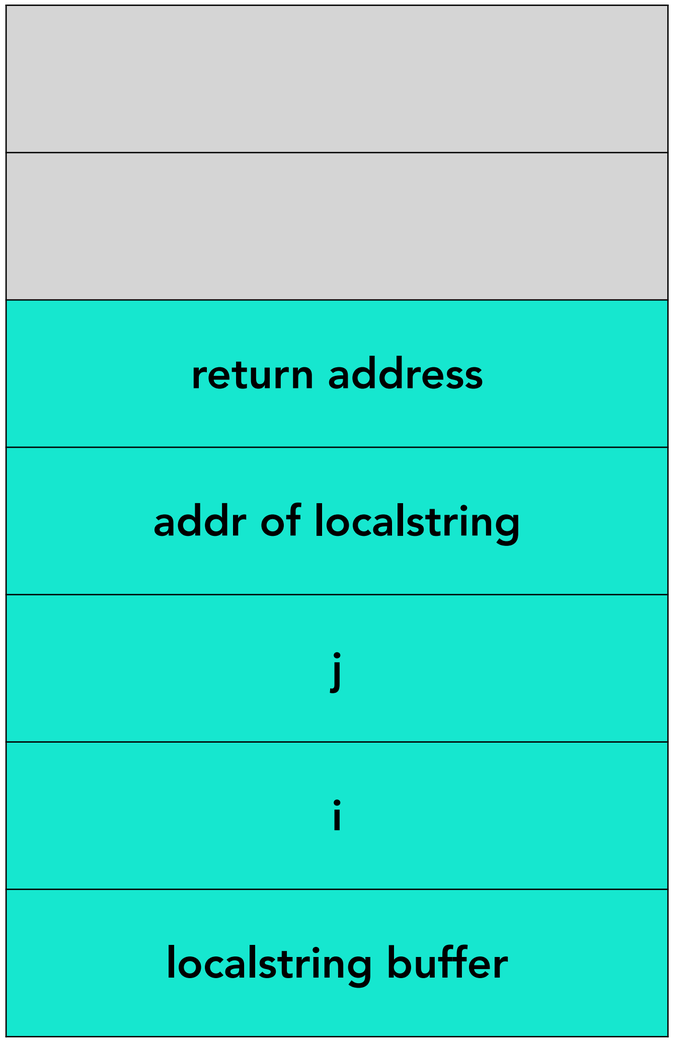
```
void f() {
    char localstring[80] =
"x10\x01\x48\x08_%08x.%08x.|
%s|";
    int i, j;
    printf(localstring);
}
```



Format string vulnerabilities

Reading arbitrary memory

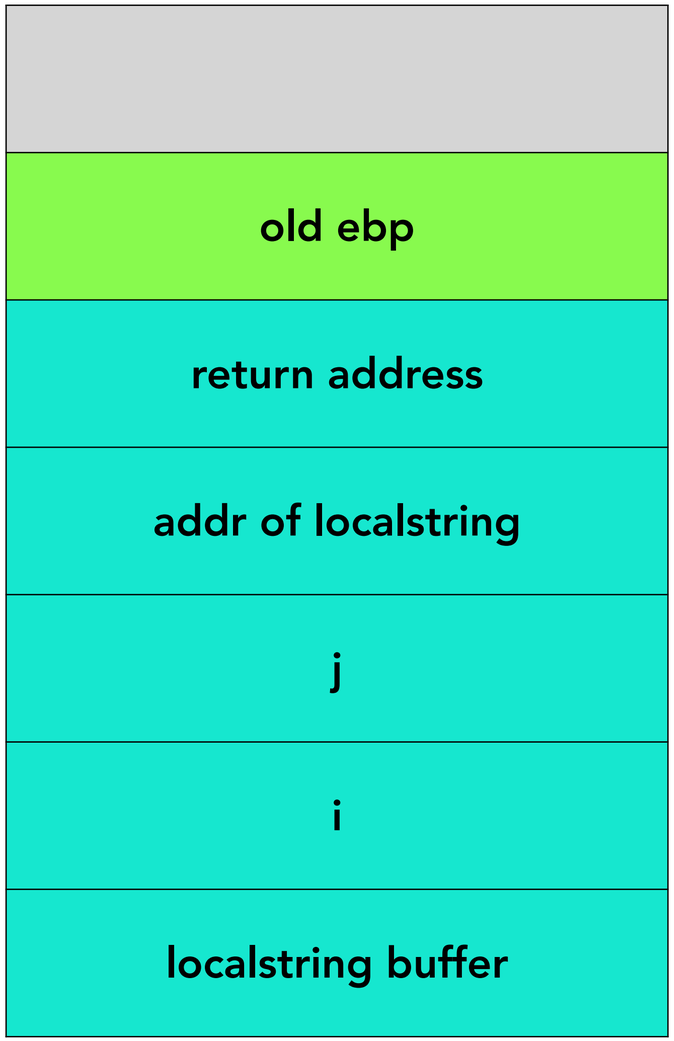
```
void f() {
    char localstring[80] =
"x10\x01\x48\x08_%08x.%08x.|
%s|";
    int i, j;
    printf(localstring);
}
```



Format string vulnerabilities

Reading arbitrary memory

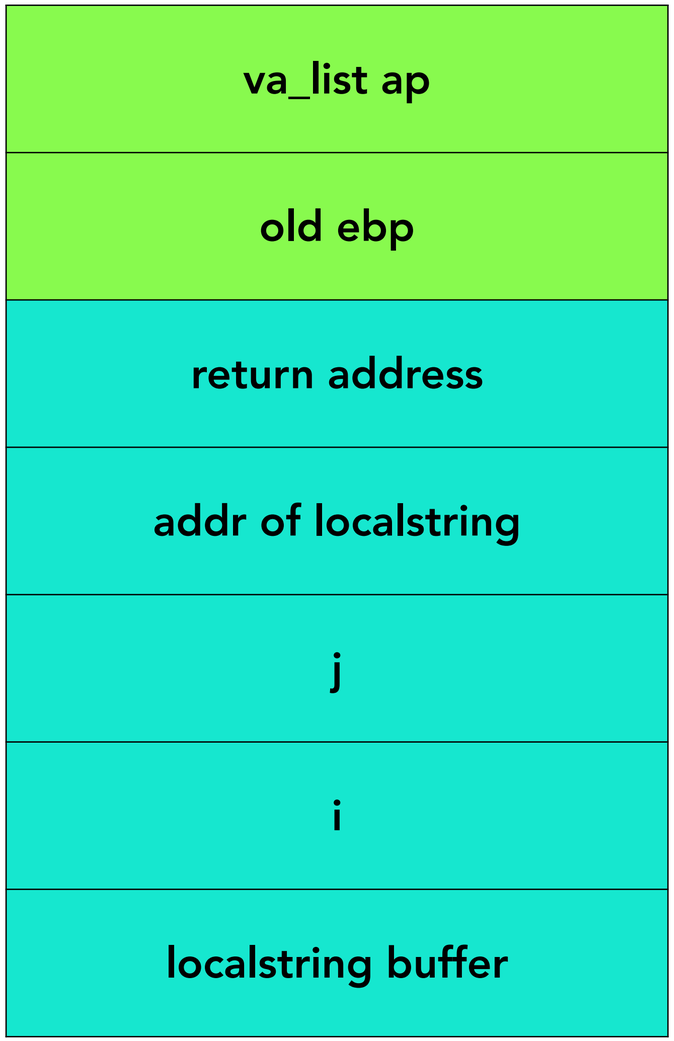
```
void f() {
    char localstring[80] =
"x10\x01\x48\x08_%08x.%08x.|
%s|";
    int i, j;
    printf(localstring);
}
```



Format string vulnerabilities

Reading arbitrary memory

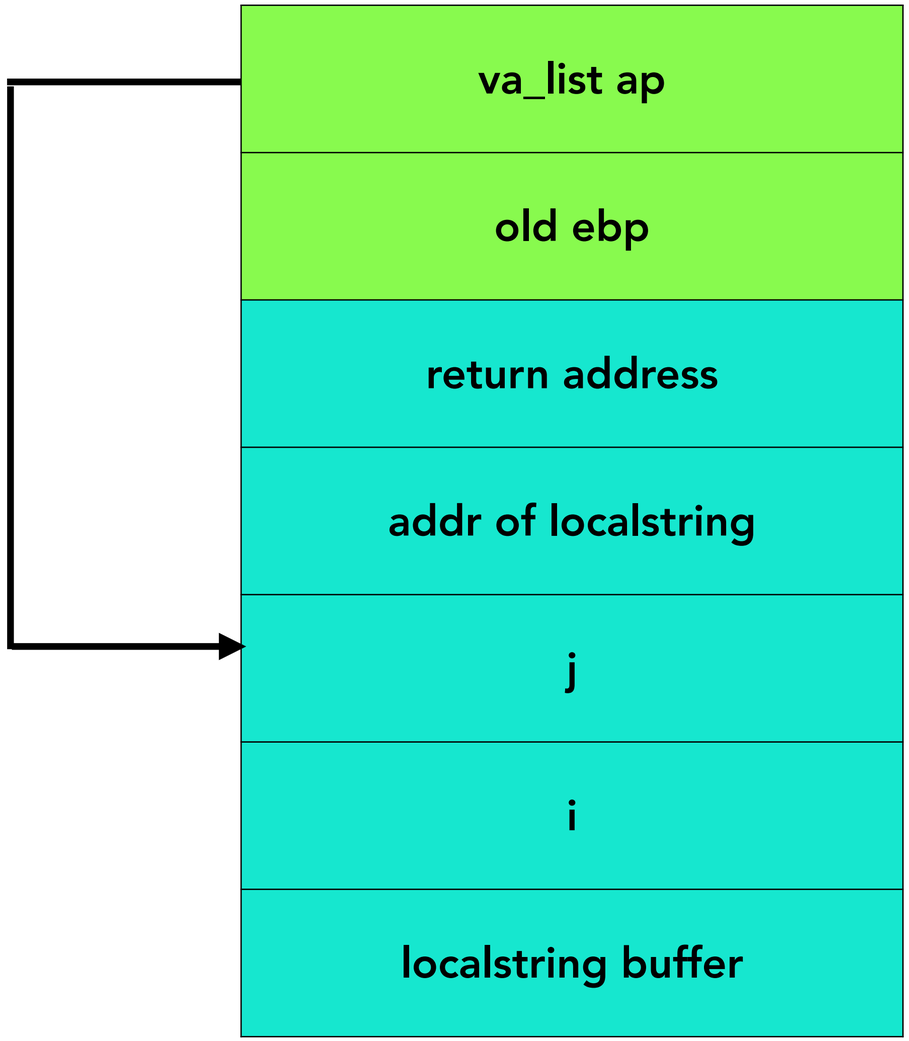
```
void f() {
    char localstring[80] =
"x10\x01\x48\x08_%08x.%08x.|
%s|";
    int i, j;
    printf(localstring);
}
```



Format string vulnerabilities

Reading arbitrary memory

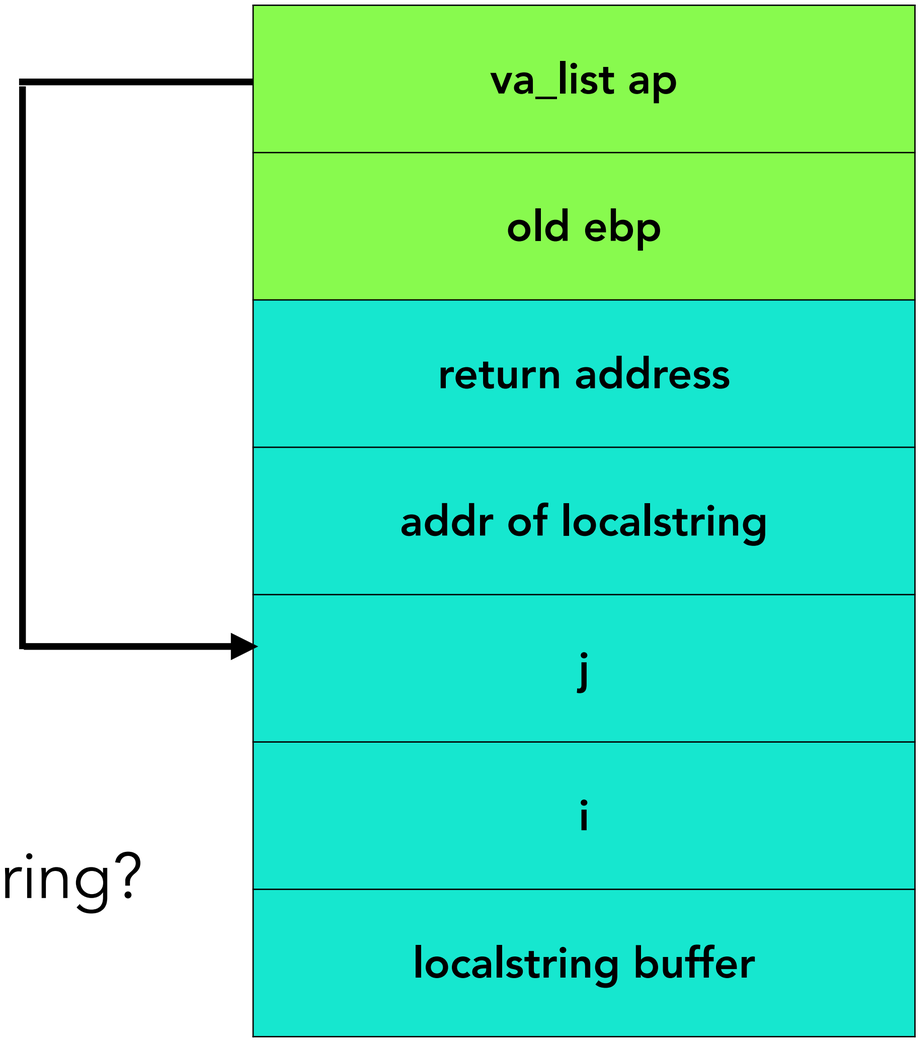
```
void f() {
    char localstring[80] =
"x10\x01\x48\x08_%08x.%08x.|
%s|";
    int i, j;
    printf(localstring);
}
```



Format string vulnerabilities

Reading arbitrary memory

```
void f() {
    char localstring[80] =
"x10\x01\x48\x08_%08x.%08x.|
%s|";
    int i, j;
    printf(localstring);
}
```

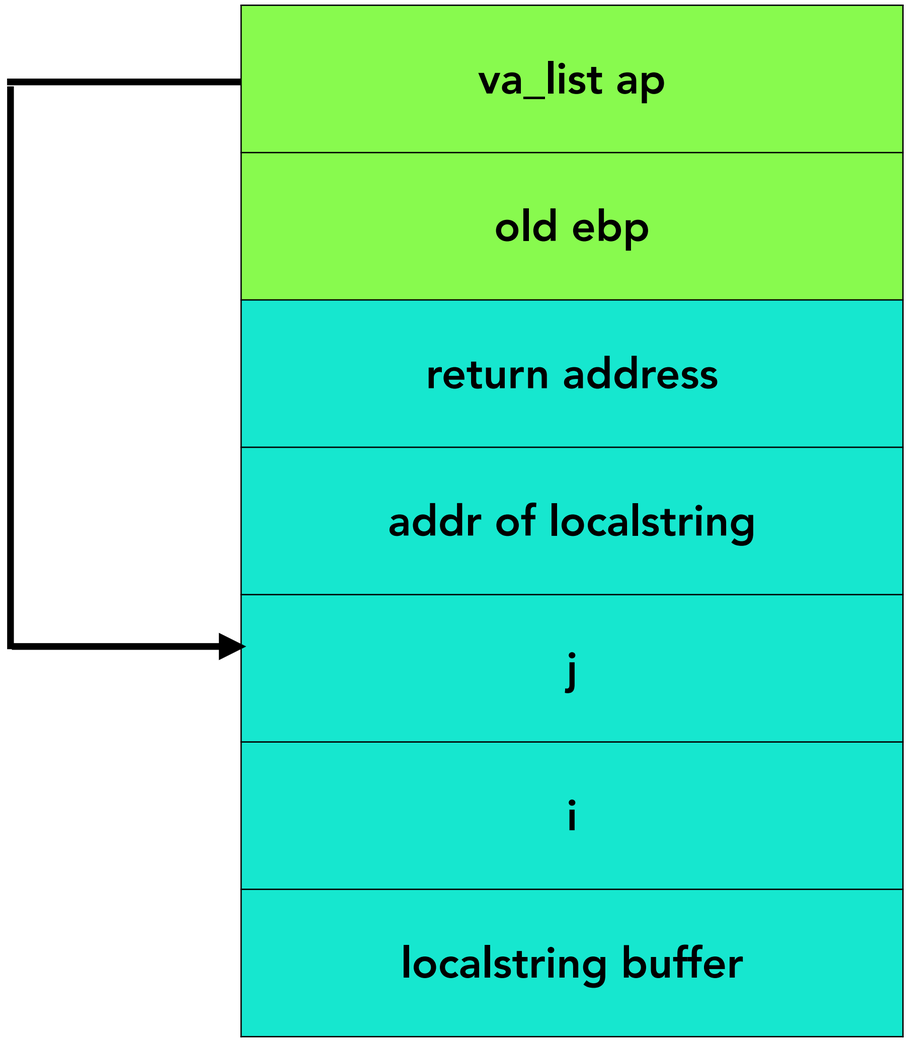


How is `printf` going to parse this format string?

Format string vulnerabilities

Reading arbitrary memory

```
void f() {
    char localstring[80] =
"x10\x01\x48\x08 %08x.
%08x.|%s|";
    int i, j;
    printf(localstring);
}
```



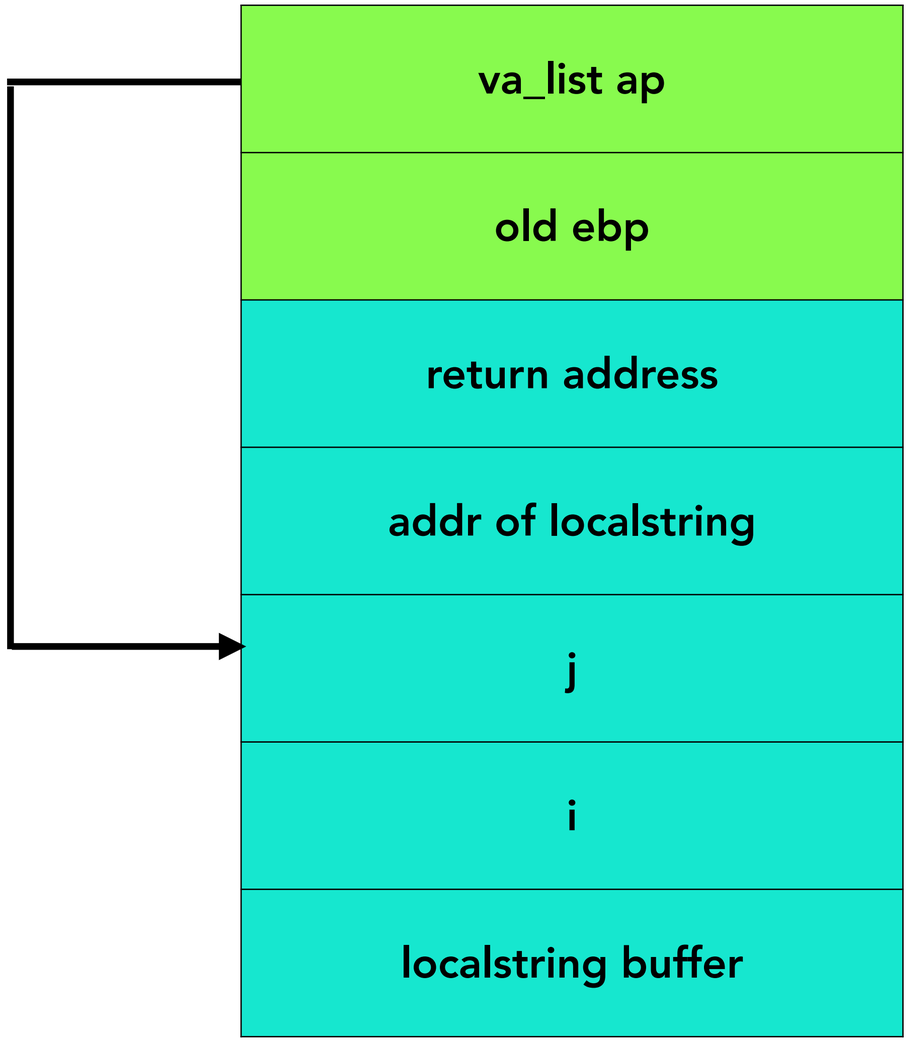
These are just some unprintable bytes (they encode `0x08480110`... important later)

Format string vulnerabilities

Reading arbitrary memory

```
void f() {
    char localstring[80] =
"x10\x01\x48\x08_%08x.
%08x.|%s|";
    int i, j;
    printf(localstring);
}
```

This is an underscore

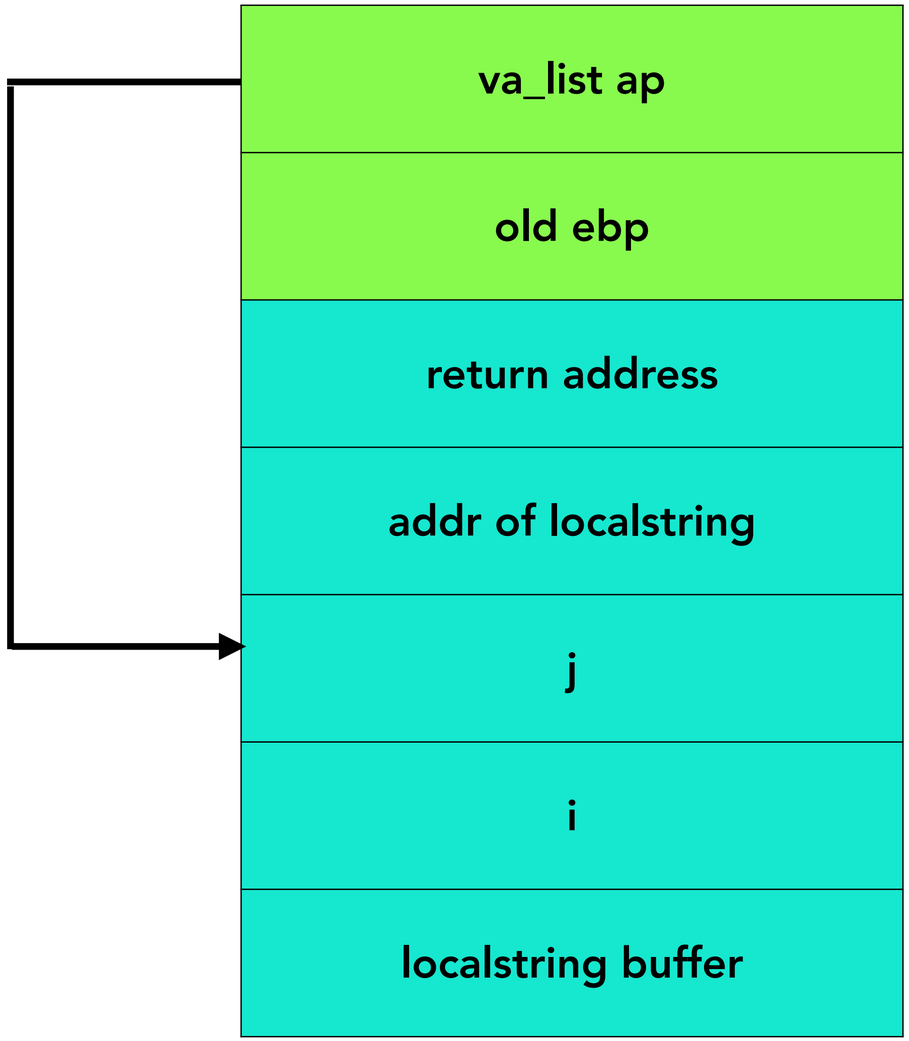


Format string vulnerabilities

Reading arbitrary memory

```
void f() {
    char localstring[80] =
"x10\x01\x48\x08_%08x.
%08x.|%s|";
    int i, j;
    printf(localstring);
}
```

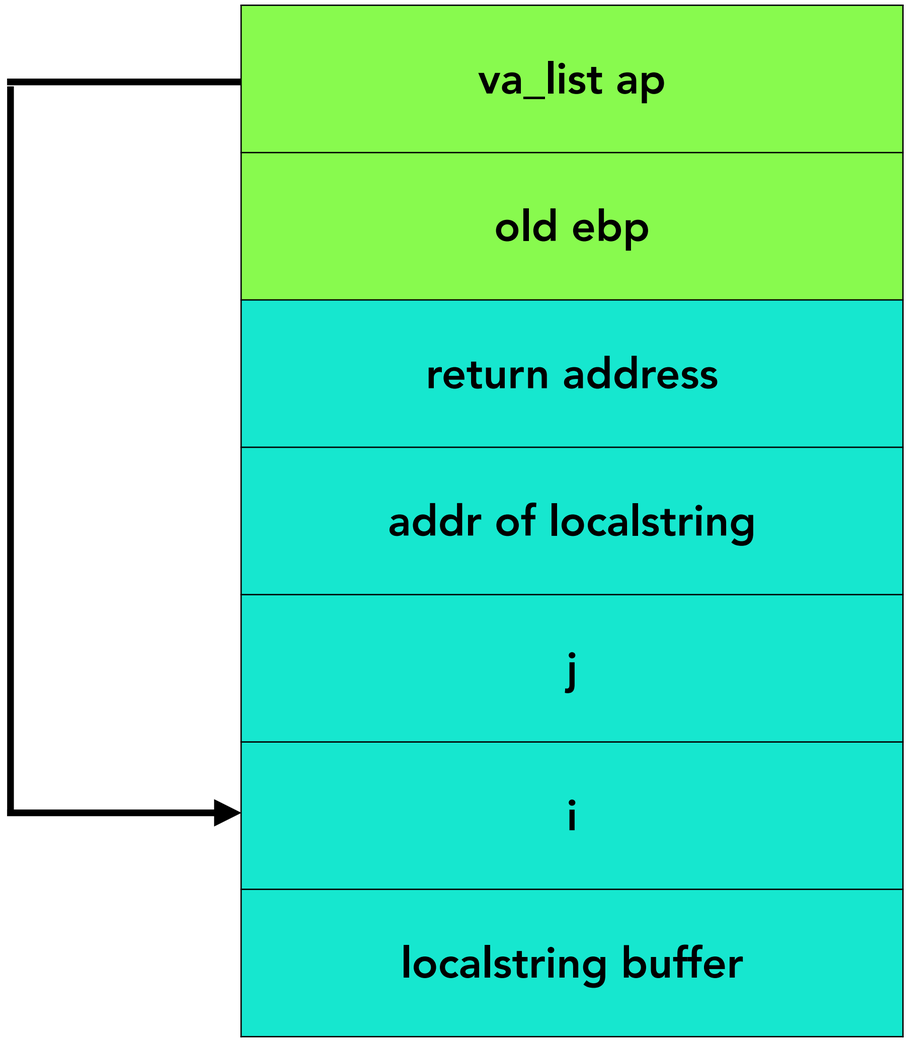
What does this specifier do?



Format string vulnerabilities

Reading arbitrary memory

```
void f() {
    char localstring[80] =
"x10\x01\x48\x08_%08x.
%08x.|%s|";
    int i, j;
    printf(localstring);
}
```



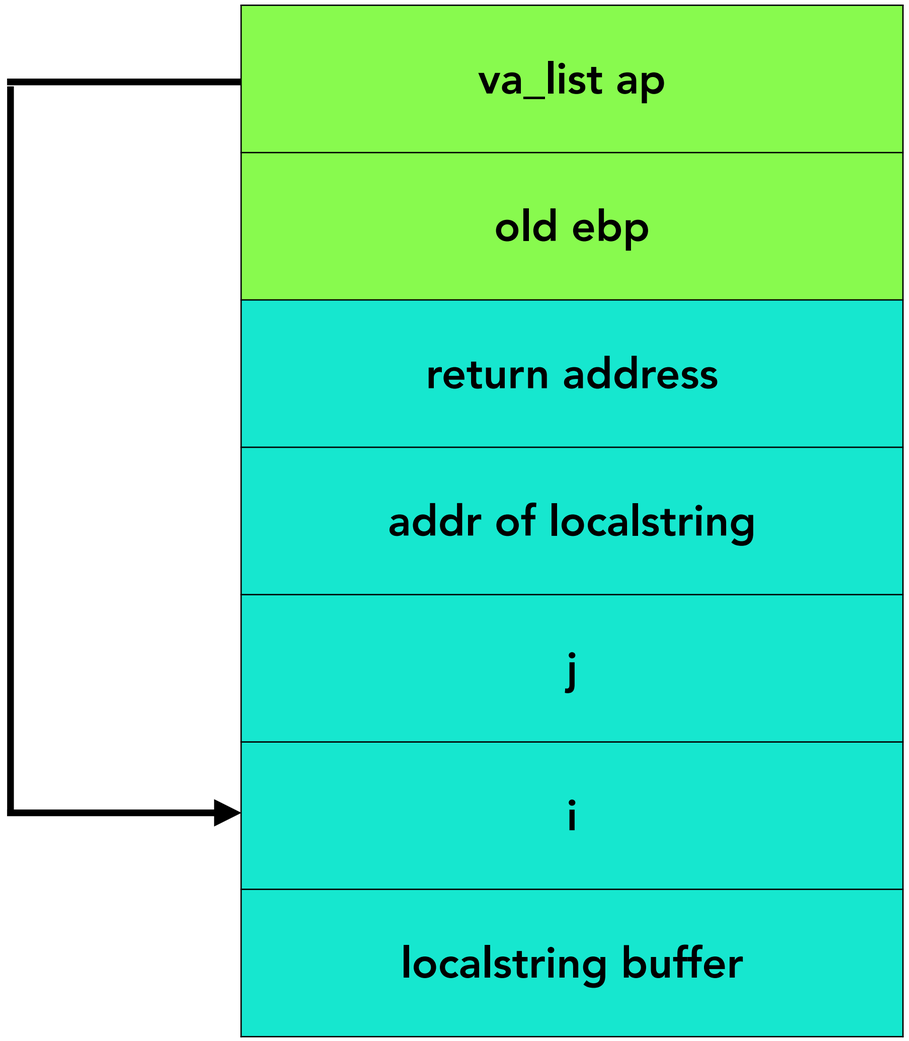
Print unsigned hex integer… therefore moving ap

Format string vulnerabilities

Reading arbitrary memory

```
void f() {
    char localstring[80] =
"x10\x01\x48\x08_%08x.
%08x.|%s|";
    int i, j;
    printf(localstring);
}
```

This is a period

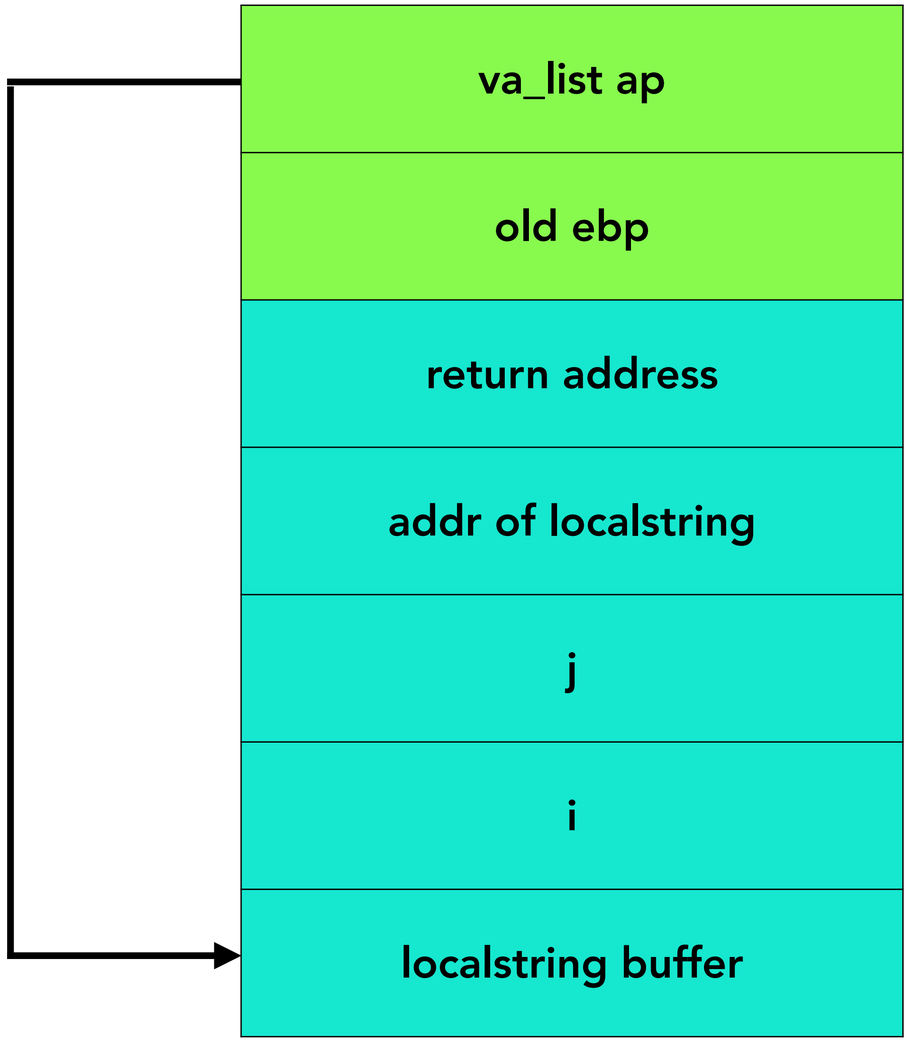


Format string vulnerabilities

Reading arbitrary memory

```
void f() {
    char localstring[80] =
"x10\x01\x48\x08_%08x.
%08x.|%s|";
    int i, j;
    printf(localstring);
}
```

Again, print unsigned hex integer and move ap, plus a period

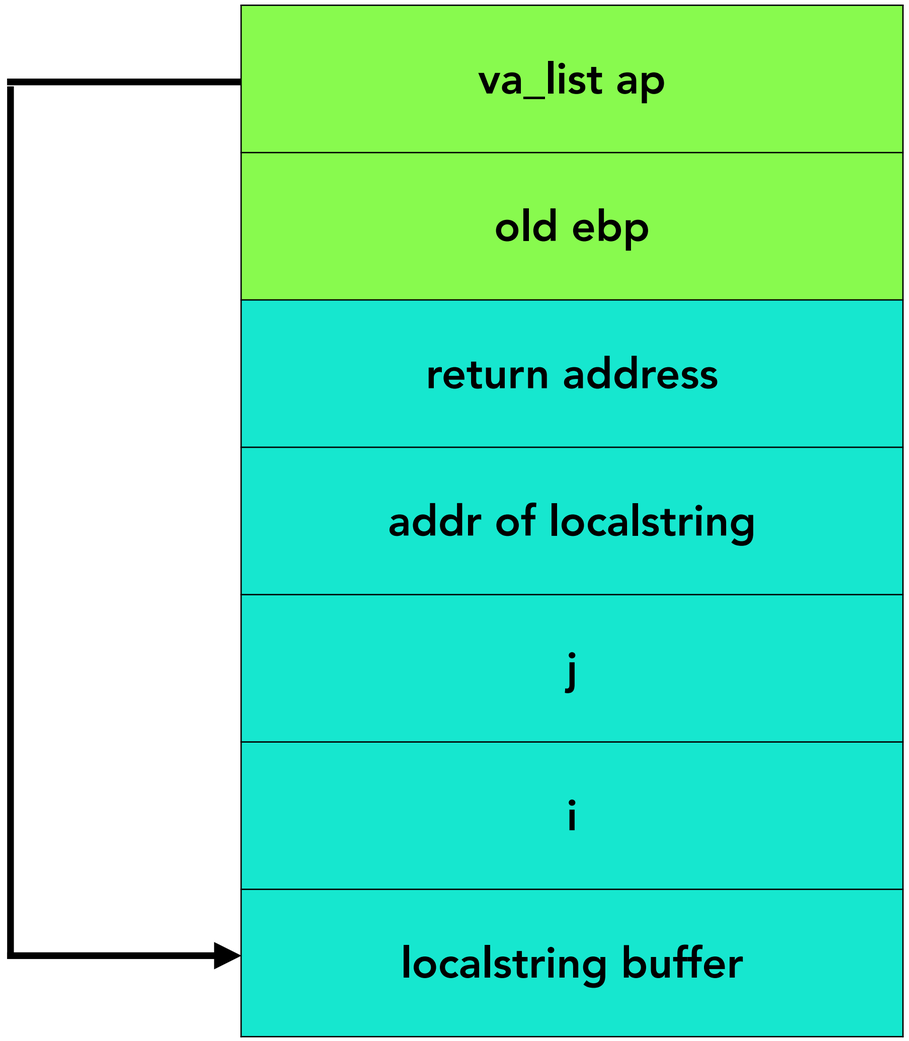


Format string vulnerabilities

Reading arbitrary memory

```
void f() {
    char localstring[80] =
"x10\x01\x48\x08_%08x.
%08x.|%s|";
    int i, j;
    printf(localstring);
}
```

Where is the argument pointer pointing to now?

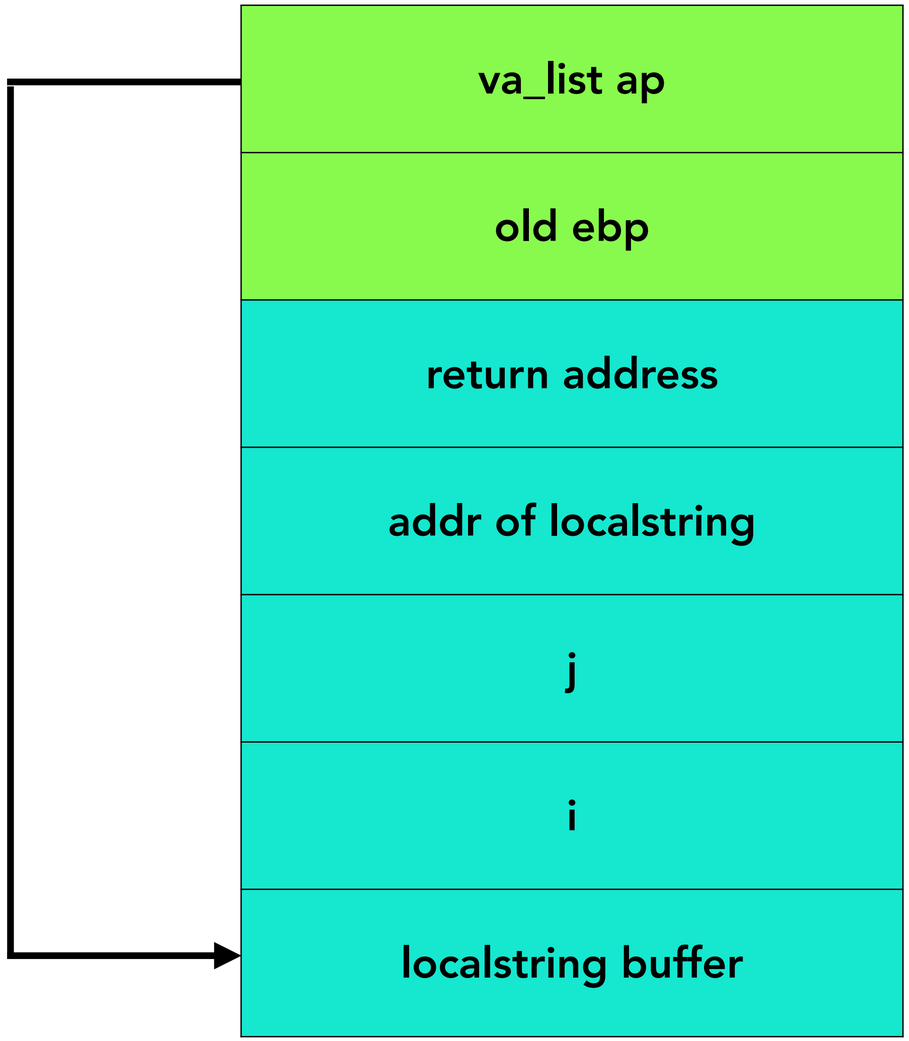


Format string vulnerabilities

Reading arbitrary memory

```
void f() {
    char localstring[80] =
"x10\x01\x48\x08_%08x.
%08x.|%s|";
    int i, j;
    printf(localstring);
}
```

What does this specifier do?



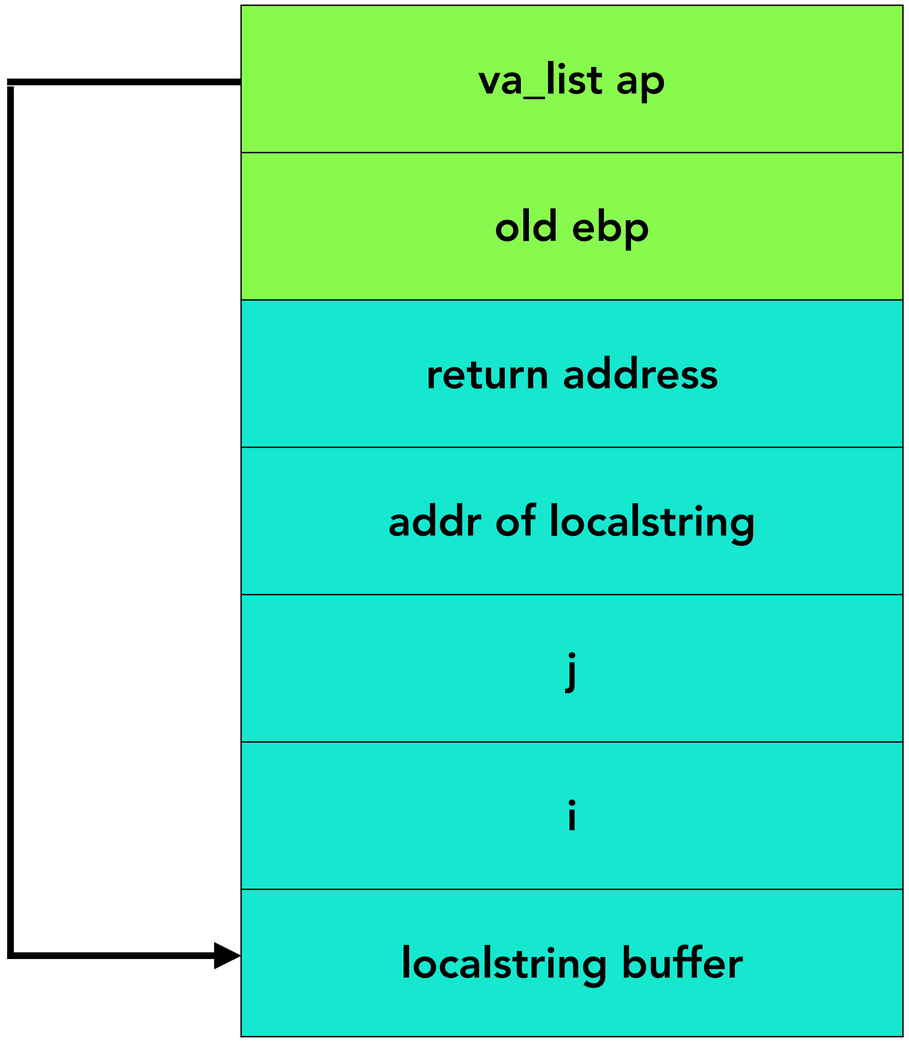
Format string vulnerabilities

Reading arbitrary memory

```
void f() {
    char localstring[80] =
"x10\x01\x48\x08_%08x.
%08x.|%s|";
    int i, j;
    printf(localstring);
}
```

This will read memory at location 0x08480110!

by extension, can perform any arbitrary read of process memory!



But it gets worse!

`printf` can write, too!

- My favorite specifier, %n
  - *"Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location."*

But it gets worse!

`printf` can write, too!

- My favorite specifier, %n
  - *"Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location."*

```
int x = 0;
printf("Hello %n", &x);
```

But it gets worse!

`printf` can write, too!

- My favorite specifier, %n
  - *"Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location."*

```
int x = 0;
printf("Hello %n", &x);
```

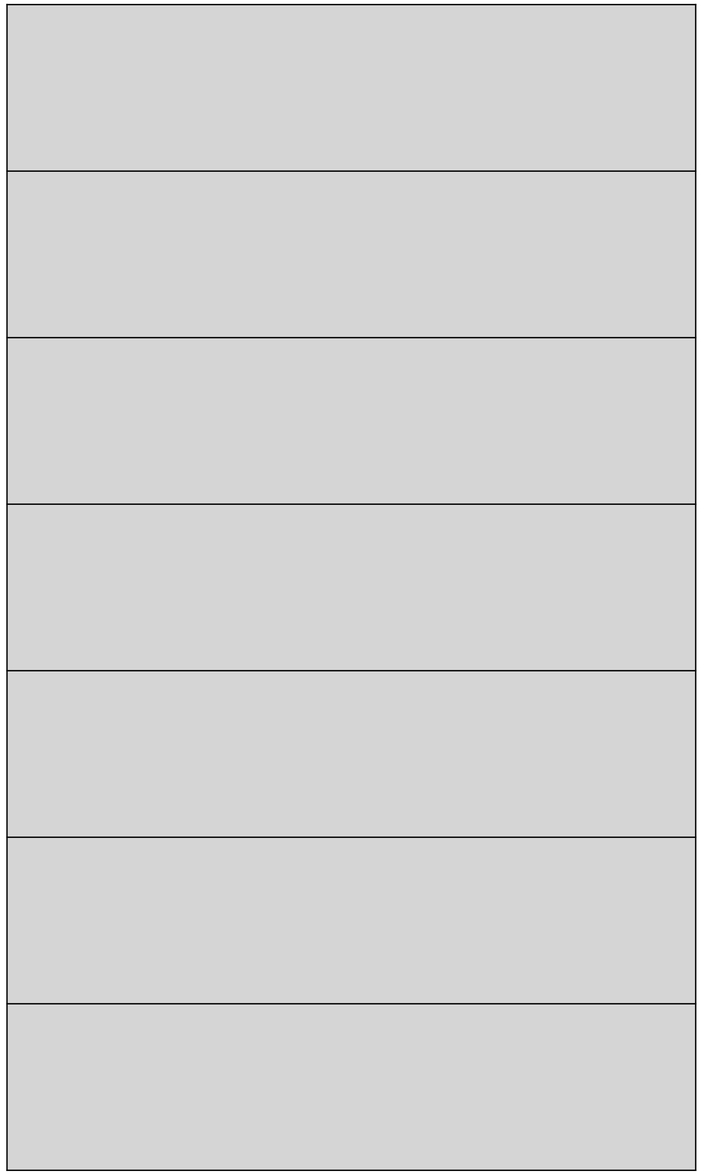
  - After this code, the value of x will be 6.

How can we exploit this?

Overwriting return address w/ `printf`

```
void f(){
    char buf[200];
    strncpy(buf, UserGeneratedString(), 200);
    printf(buf);
}

void main(){
    f()
}
```

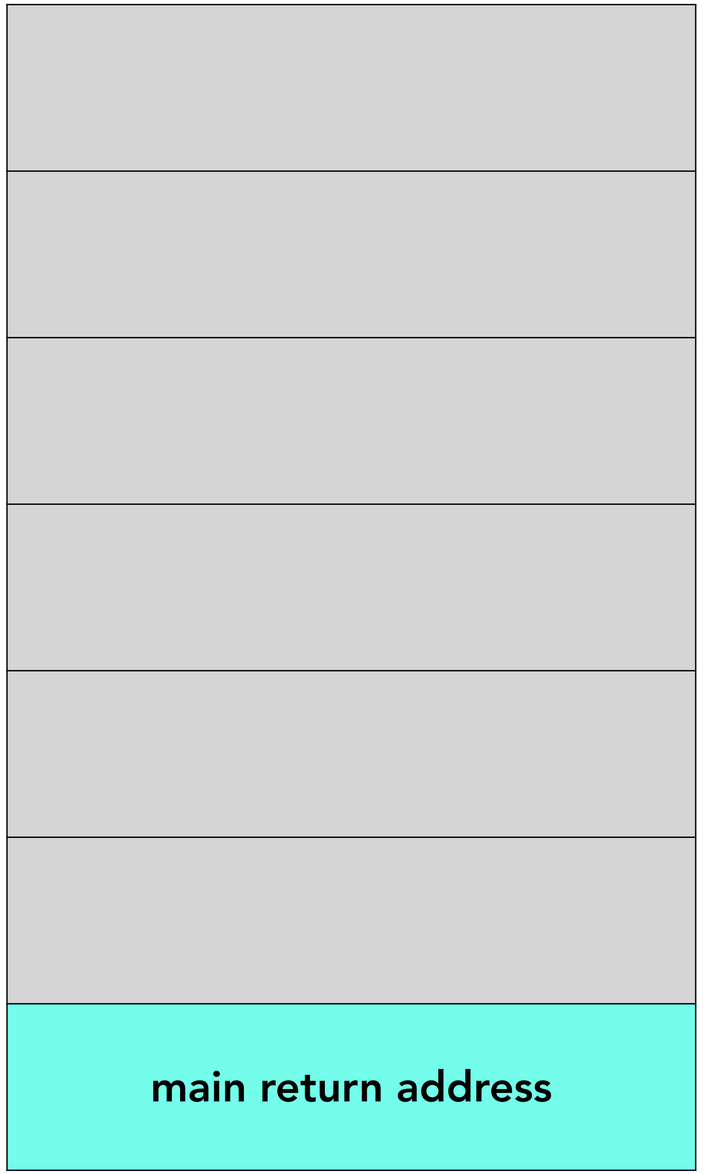


How can we exploit this?

Overwriting return address w/ `printf`

```
void f(){
    char buf[200];
    strncpy(buf, UserGeneratedString(), 200);
    printf(buf);
}

void main(){
    f()
}
```

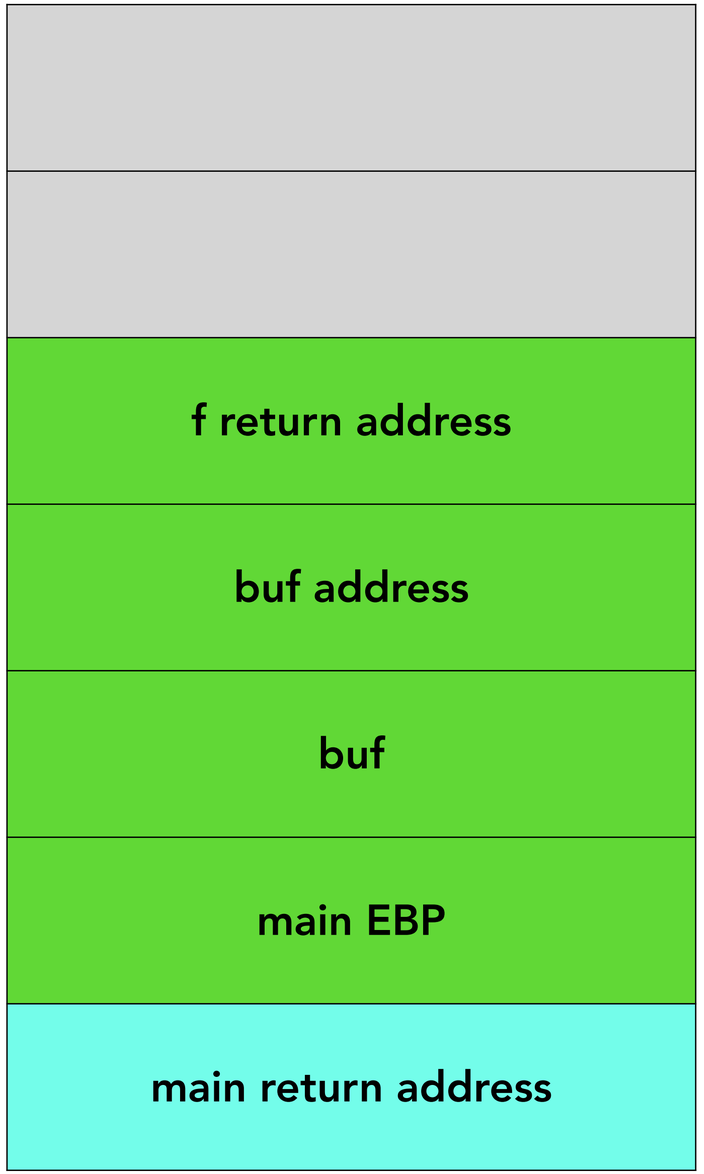


How can we exploit this?

Overwriting return address w/ `printf`

```
void f(){
    char buf[200];
    strncpy(buf, UserGeneratedString(), 200);
    printf(buf);
}

void main(){
    f()
}
```

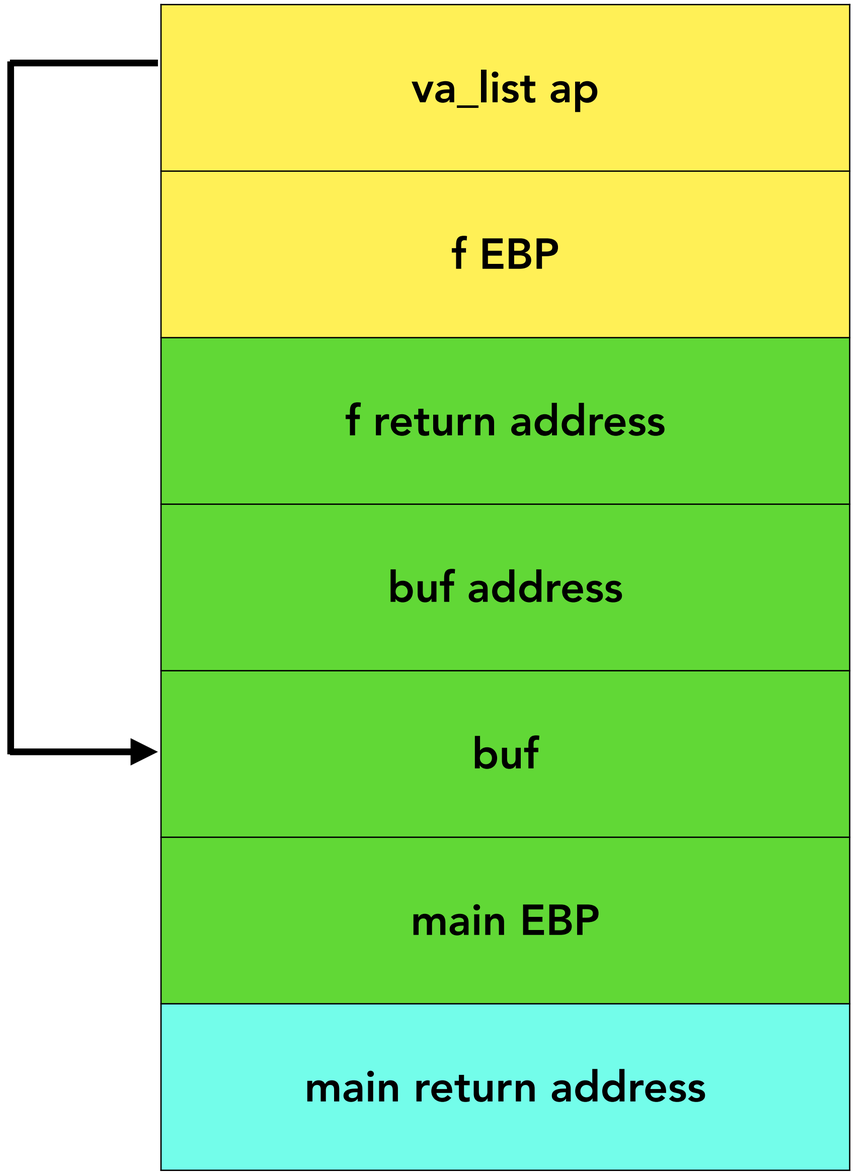


How can we exploit this?

Overwriting return address w/ `printf`

```
void f(){
    char buf[200];
    strncpy(buf, UserGeneratedString(), 200);
    printf(buf);
}

void main(){
    f()
}
```



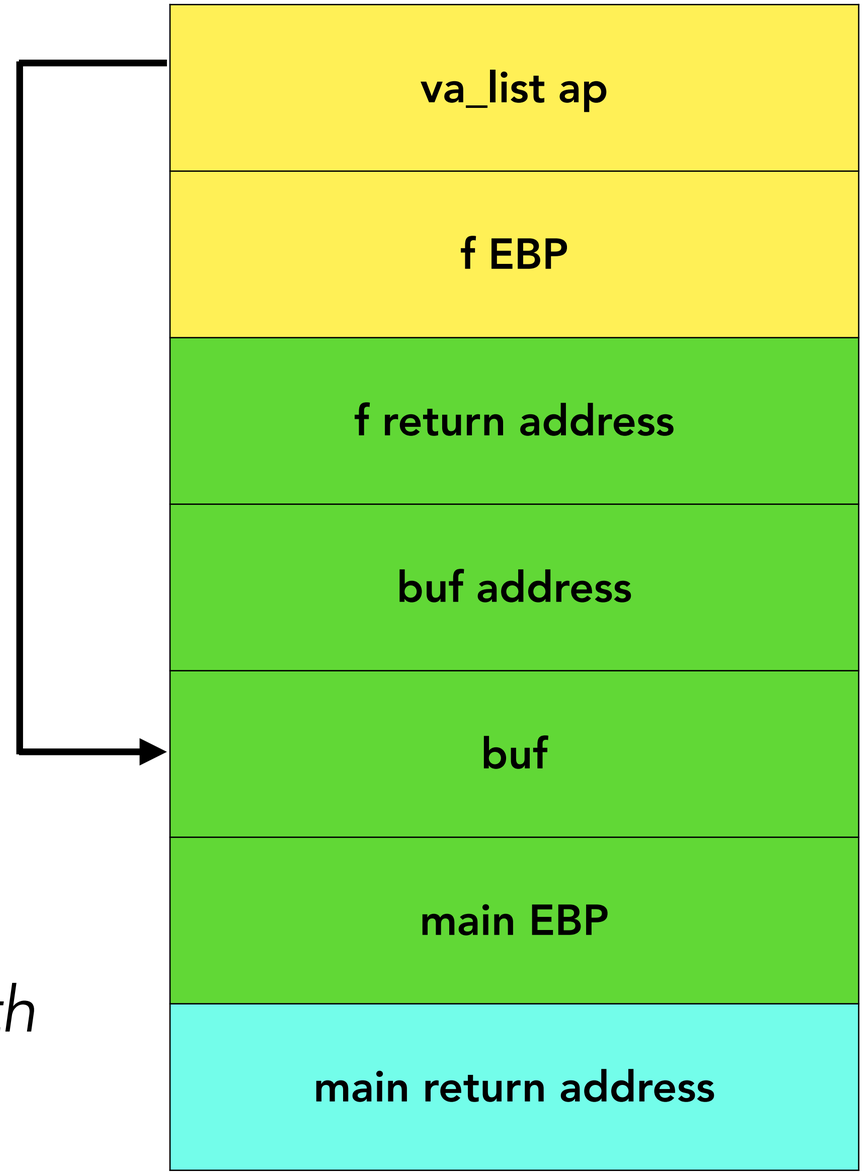
What can we do from here?

How can we exploit this?

Overwriting return address w/ `printf`

```
void f(){
    char buf[200];
    strncpy(buf, UserGeneratedString(), 200);
    printf(buf);
}

void main(){
    f()
}
```



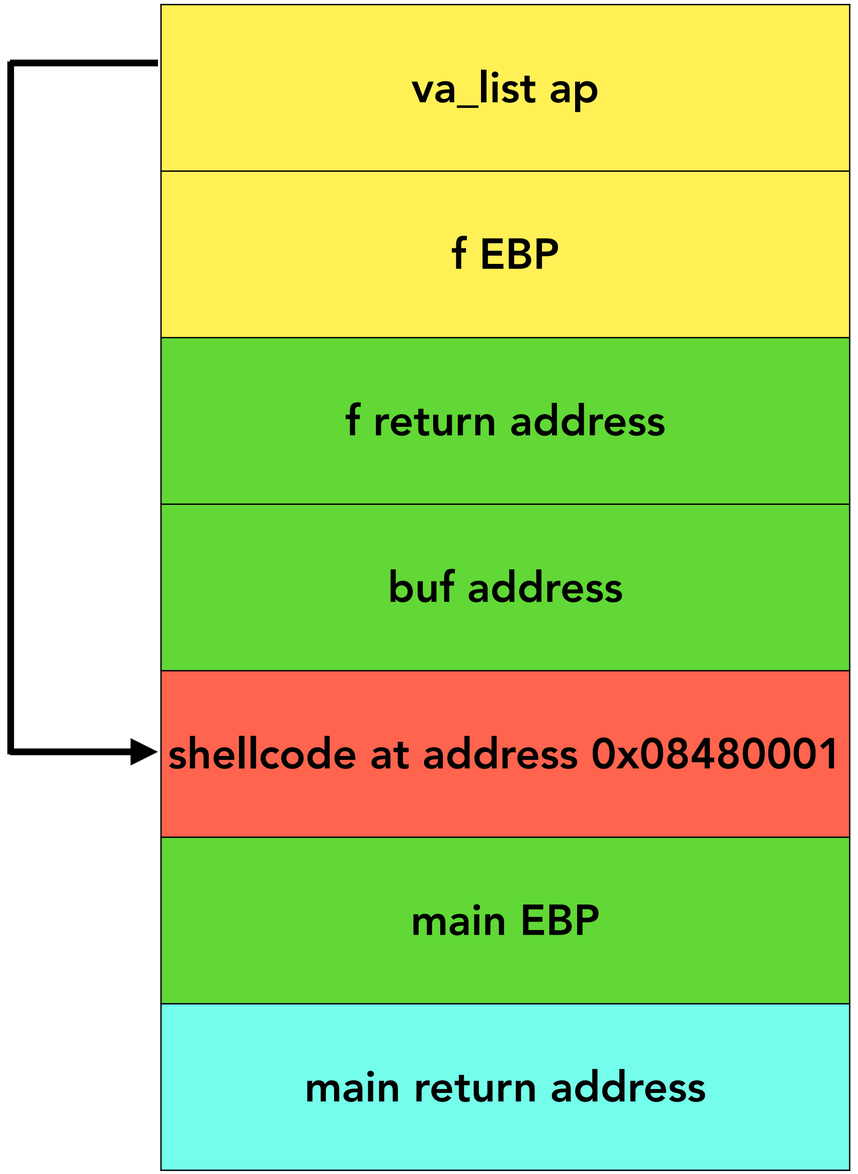
If UserGeneratedString() contains %n… we can write the return address one byte at a time based on length of our string

How can we exploit this?

Overwriting return address w/ `printf`

```
void f(){
    char buf[200];
    strncpy(buf, UserGeneratedString(), 200);
    printf(buf);
}

void main(){
    f()
}
```



Overwrite return address to point to buffer, which contains shellcode!

game over, gg

`printf` %n tricks

- Seems kinda hard…
- Specify width
  - %n will account for “padding bytes” specified in the width parameter, for example:
    - `printf(“%5d”, 10)` will print 3 spaces followed by the integer: “ 10”; %n will return 5 (not 2)
- Chain together reads and writes
  - Use arbitrary reads to know where buffer is, then construct payload to be appropriate length to write to return address
- **Note: %n is considered dangerous**. Most libraries don’t support it.

What do I do about `printf`?

- If you find yourself needing to use `printf`…
  - Be careful (check, double check, triple check your calls to printf)
- Sanitize user inputs (do you really want users inputting %s into the string?)
- `snprintf(str, size, format, …);`
  - Only prints *size* characters (and so can restrict arbitrary memory reads)

So, in conclusion…

- Functions that take format strings act like little command interpreters…
  - Anything that can step outside of the semantics of your runtime system like this is **potentially dangerous**
- Letting attackers decide which commands to pass into your command interpreter is… a bad idea
  - Deepak's version: **Don't let the attacker program the weird machine.**

Integer Overflow Vulnerabilities

Pop quiz!

```
a = 100;
b = 200;
c = a + b;

printf("%d %d %d\n", a, b, c);
```

What will the code produce?

Pop quiz!

```
a = 100;
b = 200;
c = a + b;

printf("%d %d %d\n", a, b, c);
```

What will the code produce?

It depends…

Pop quiz!

```
int a = 100;
int b = 200;
int c = a + b;

printf("%d %d %d\n", a, b, c);
```

100, 200, 300

Pop quiz

```
int a = 100;
char b = 200;
int c = a + b;

printf("%d %d %d\n", a, b, c);
```

100, -54, 44

Integer overflow / underflow

- C defines fixed-width integer types (`short`, `int`, `long`, etc.) that do not always behave like the integers you might recall from… well, math
- Because of the fixed width, it is possible to *overflow* or *wrap* maximum expressible number for the type used
  - Or *underflow* in the case of negative numbers

Who cares about numbers?

- What could go wrong here?
- What if **n** is too large?
- What if **n** is negative?

```
my_type* foo(int n)
{
    my_type *ptr = malloc(n *
sizeof(my_type));
    for(int i = 0; i < n; i++) {
        memset(&ptr[i], i,
sizeof(my_type));
    }
 return ptr;
}
```

Type conversations are a nightmare

- Integer type conversions are a **very** common source of security vulnerabilities
  - What's a type conversion?

Type conversations are a nightmare

- Integer type conversions are a **very** common source of security vulnerabilities
  - What's a type conversion?
- Conversions happen in three ways
  - Truncation
  - Zero-extension
  - Sign-extension

A quick review —

- `char`
  - At least 8 bits. `sizeof(char) == 1`
- `short`
  - At least 16 bits
- `int`
  - Natural word size of the architecture, at least 16 bits
- `long`
  - At least 32 bits

Truncation

- Truncation happens when a value with a wider type is converted to a narrower type
- When the value is truncated, *high-order bytes are removed* so it can be the same width as the narrower type

```
uint32_t i = 0xDEADBEEF;
uint16_t j = i;

// j = 0xBEEF
```

Zero-extension

- Zero-extension occurs when a value with a narrower, *unsigned* type is converted to a wider type
- When a value is zero-extended, it is widened so that it is the same width as the wider type; the new bytes are unset (0)

```
uint16_t i = 0xBEEF;
uint32_t j = i;

// j = 0x0000BEEF
```

Sign-extension

- Sign extension occurs when a value with a narrower, signed type is converted to a wider type
- When a value is sign-extended, it is widened so that it is the same width as the wide type
  - If the sign bit of the original value is set, the new bytes are set
  - If the sign bit is unset, the new bytes are unset

```
int8_t i = 127;          // 0111 1111
int8_t j = -127;         // 1000 0001 (2s complement)
int16_t k_i = i;         // 0000 0000 0111 1111
int16_t j_i = j;         // 1111 1111 1000 0001
```

When do we do what conversion?

- I'm not testing this, but just so you know…

From	To	Method	Lost or Misinterpreted
signed char	short int	Sign-extend	Safe
signed char	long int	Sign-extend	Safe
signed char	unsigned char	Preserve bit pattern; high-order bit no longer sign bit	Misinterpreted
signed char	unsigned short int	Sign-extend to short; convert short to unsigned short	Lost
signed char	unsigned long int	Sign-extend to long; convert long to unsigned long	Lost
signed short int	char	Truncate to low-order byte	Lost
signed short int	long int	Sign-extend	Safe
signed short int	unsigned char	Truncate to low-order byte	Lost
signed short int	unsigned short int	Preserve bit pattern; high-order bit no longer sign bit	Misinterpreted
signed short int	unsigned long int	Sign extend to long; convert long to unsigned	Lost
signed long int	char	Truncate to low order byte	Lost
signed long int	short int	Truncate to low order bytes	Lost
signed long int	unsigned char	Truncate to low order byte	Lost
signed long int	unsigned short int	Truncate to low order bytes	Lost

When do we do what conversion?

- I'm not testing this, but just so you know…

From	To	Method	Lost or Misinterpreted
unsigned char	signed char	Preserve bit pattern; high-order bit becomes sign bit	Misinterpreted
unsigned char	signed short int	Zero-extend	Safe
unsigned char	signed long int	Zero-extend	Safe
unsigned char	unsigned short int	Zero-extend	Safe
unsigned char	unsigned long int	Zero-extend	Safe
unsigned short int	signed char	Preserve low-order byte	Lost
unsigned short int	signed short int	Preserve bit pattern; high-order bit becomes sign bit	Misinterpreted
unsigned short int	signed long int	Zero-extend	Safe
unsigned short int	unsigned char	Preserve low-order byte	Lost
unsigned long int	signed char	Preserve low-order byte	Lost
unsigned long int	signed short int	Preserve low-order word	Lost
unsigned long int	signed long int	Preserve bit pattern; high-order bit becomes sign bit	Misinterpreted
unsigned long int	unsigned char	Preserve low-order byte	Lost
unsigned long int	unsigned short int	Preserve low-order word	Lost

Type conversions happen all the time

- Explicit
  - `int i = (int) 4.5;`
- Implicit
  - `signed char i = 1; // assignment conversion`
  - `unsigned int j = 2; //assignment conversion`
  - `if (i < j) {…} // comparison conversion`
  - ```
    void function (int arg);
    function(5.3); // function argument conversion
    ```
- Conversion rules are **complex**, but ever present :)
    ```

Example of integers causing problems

```
struct sockaddr_in
{
    short sin_family;
    u_short sin_port;
};

sockaddr_in sockaddr;
int port; // Get this from a user

if (port < 1024 && !is_root) {
    // Quit, or handle error
} else {
    sockaddr.sin_port = port;
}
```

What's wrong with this code?

Example of integers causing problems

```
struct sockaddr_in
{
    short sin_family;
    u_short sin_port;
};

sockaddr_in sockaddr;
int port; // Get this from a user

if (port < 1024 && !is_root) {
    // Quit, or handle error
} else {
    sockaddr.sin_port = port;
}
```

- The field `sin_port` is declared as a 16-bit unsigned integer
- The variable `port` is declared as a 32-bit signed integer
- When `sin_port` is set to `port`, the two **high-order bytes of value** are truncated and the `port` number is changed

Example of integers causing problems

```
struct sockaddr_in
{
    short sin_family;
    u_short sin_port;
};

sockaddr_in sockaddr;
int port; // Get this from a user

if (port < 1024 && !is_root) {
    // Quit, or handle error
} else {
    sockaddr.sin_port = port;
}
```

- Exploit
  - Set `port` = `65979`
  - Comparison will fail
  - Assignment will truncate 65979 (`0x000101BB` in hex); to `0x01BB`, or `443`
  - `443` is port for HTTPS; now attacker has privilege to read all traffic over `443` :)

Example of integers causing problems

Nasdaq had to adjust core code after Berkshire Hathaway share price high

News By Anthony Spadafora last updated May 10, 2021

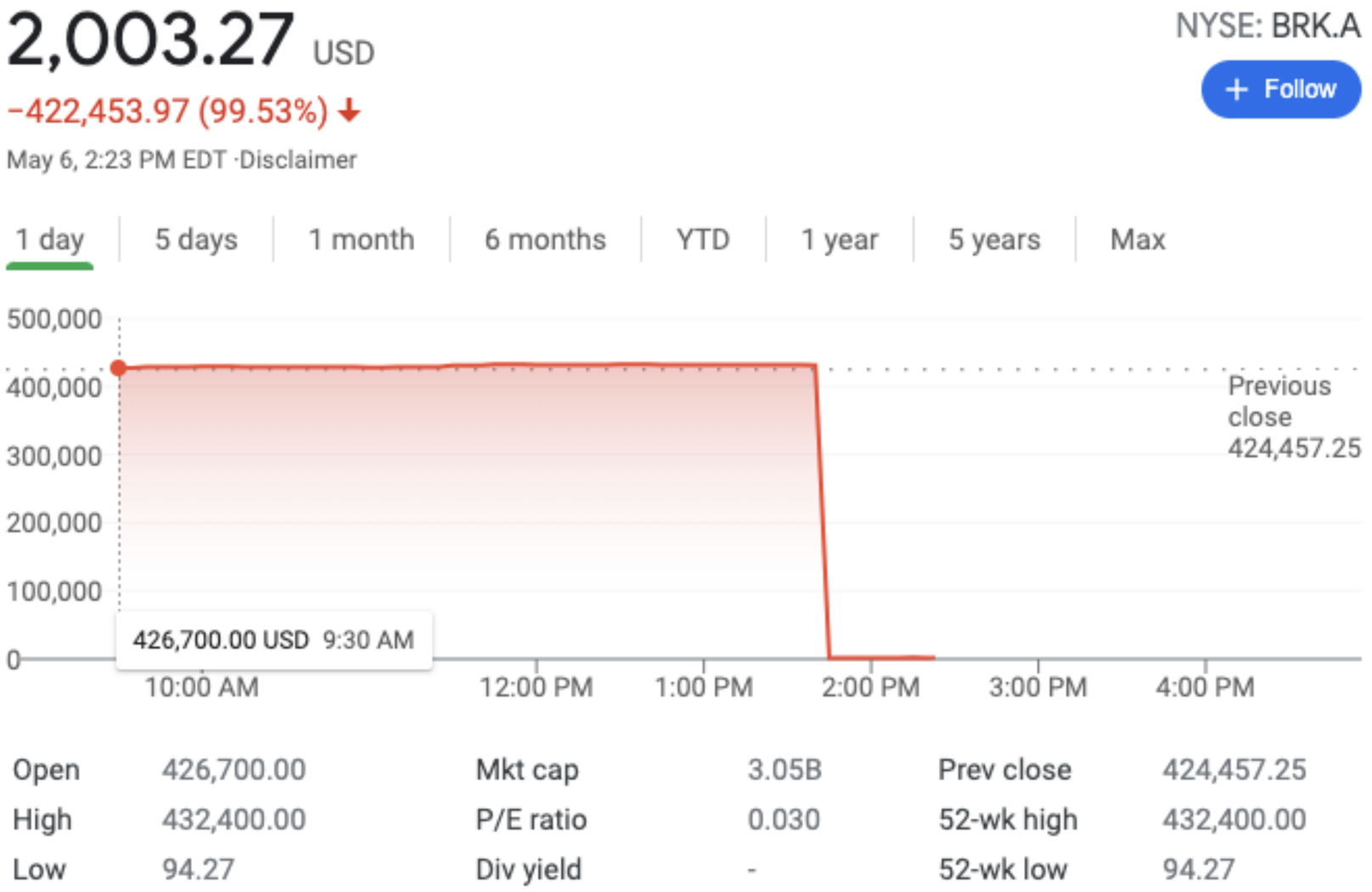
Stock price exceeded the maximum value for storing 32-bit unsigned integers

Nasdaq's computer system literally can't handle Berkshire Hathaway's sky-high stock price

By Matt Egan, CNN Business

2 min read · Updated 1:24 PM EDT, Fri May 7, 2021

Example of integers causing problems



What do I do about integers?!

- Use a **strongly typed language**
  - Essentially minimizes type conversions and is much stricter about type checking
  - Most integer overflow problems go away in Rust, Go
- Runtime checking
  - gcc -ftrapv (trap on signed overflow on add, sub, mult)
  - Safe libraries (Check out SafeInt)
- Static analysis
  - Can check code retroactively for potentially weird behavior

Next time…

- Final application security lecture!
- We talk about defenses and my favorite type of appsec attack, return-oriented programming :)