Memory Safety

Low-level attacks enabled by a lack of **Memory Safety**

A memory safe program execution:

1. only **creates pointers** through **standard means**
   - `p = malloc(…)`, or `p = &x`, or `p = &buf[5]`, etc.
2. only uses a pointer to **access memory** that **"belongs" to that pointer**

Combines two ideas:

temporal safety and **spatial safety**

Spatial safety

- View pointers as triples (**p**,***b***,***e***)
  - **p** is the actual pointer
  - ***b*** is the base of the memory region it may access
  - ***e*** is the extent (bounds) of that region
- **Access allowed** *iff* ***b*** ≤ **p** ≤ **e**-`sizeof`(`typeof`(**p**))
- Operations:
  - Pointer arithmetic increments **p**, leaves ***b*** and ***e*** alone
  - Using `&`: ***e*** determined by size of original type

Examples

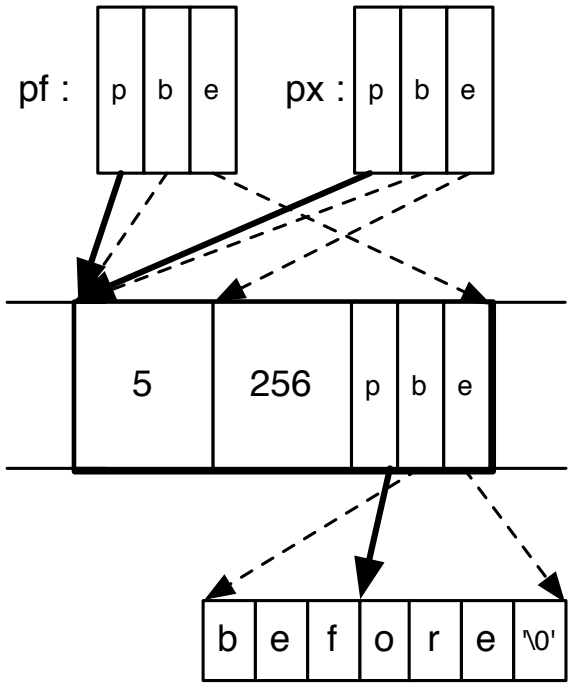
```
struct foo {
 char buf[4];
 int x;
};
```

```
int x;          // assume sizeof(int)=4
int *y = &x;    // p = &x, b = &x, e = &x+4
int *z = y+1;   // p = &x+4, b = &x, e = &x+4
*y = 3;         // OK: &x ≤ &x ≤ (&x+4)-4
*z = 3;         // Bad: &x ≤ &x+4 ≰ (&x+4)-4
```

```
struct foo f = { "cat", 5 };
char *y = &f.buf; // p = b = &f.buf, e = &f.buf+4
y[3] = 's';    // OK: p = &f.buf+3 ≤ (&f.buf+4)-1
y[4] = 'y';    // Bad: p = &f.buf+4 ≰ (&f.buf+4)-1
```

Visualized example

```
struct foo {
  int x;
  int y;
  char *pc;
};
struct foo *pf = malloc(...);
pf->x = 5;
pf->y = 256;
pf->pc = "before";
pf->pc += 3;
int *px = &pf->x;
```



No buffer overflows

- A buffer overflow violates spatial safety

```
void copy(char *src, char *dst, int len)
{
  int i;
  for (i=0;i<len;i++) {
    *dst = *src;
    src++;
    dst++;
  }
}
```

  - Overrunning the bounds of the source and/or destination buffers implies either **`src`** or **`dst`** is illegal

No format string attacks

- The call to **`printf`** dereferences illegal pointers

```
char *buf = “%d %d %d\n”;
printf(buf);
```

  - View the stack as a buffer defined by the number and types of the arguments it provides
  - The extra format specifiers construct pointers beyond the end of this buffer and dereference them

- Essentially a kind of buffer overflow

Temporal safety

- A **temporal safety violation** occurs when trying to **access undefined memory**
  - Spatial safety assures it was to a legal region
  - Temporal safety assures that region is still in play
- Memory regions either **defined** or **undefined**
  - Defined means allocated (and active)
  - Undefined means unallocated, uninitialized, or deallocated
- Pretend memory is infinitely large (we never reuse it)

No dangling pointers

- Accessing a freed pointer violates temporal safety

```
int *p = malloc(sizeof(int));
*p = 5;
free(p);
printf("%d\n",*p); // violation
```

  The memory dereferenced no longer belongs to p.

- Accessing uninitialized pointers is similarly not OK:

```
int *p;
*p = 5; // violation
```

Integer overflows?

- Allowed as long as they are not used to manufacture an illegal pointer

```
int f() {
  unsigned short x = 65535;
  x++; // overflows to become 0
  printf("%d\n",x); // memory safe
  char *p = malloc(x); // size-0 buffer!
  p[1] = 'a'; // violation
}
```

- Integer overflows often enable buffer overflows
  - Happens often enough we think of them independently

For **more on memory safety**, see
http://www.pl-enthusiast.net/2014/07/21/memory-safety/

Most languages memory safe

- The easiest way to avoid all of these vulnerabilities is to **use a memory safe language**
- Modern languages are memory safe
  - Java, Python, C#, Ruby
  - Haskell, Scala, Go, Objective Caml, Rust

- In fact, these **languages are type safe,** which is even **better** (more on this shortly)

Memory safety for C

- **C/C++ here to stay**. While not memory safe, you can write memory safe programs with them
  - The problem is that there is no guarantee
- Compilers could add **code to check for violations**
  - An out-of-bounds access would result in an immediate failure, like an *ArrayBoundsException* in Java
- This idea has been around for more than 20 years. **Performance has been the limiting factor**
  - Work by Jones and Kelly in 1997 adds 12x overhead
  - Valgrind memcheck adds 17x overhead

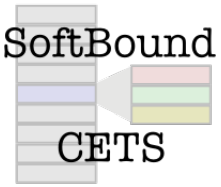
Progress

Research has been **closing the gap**

- **CCured** (2004), 1.5x slowdown
  - But no checking in libraries
  - Compiler rejects many safe programs

ccured

- **Softbound/CETS** (2010): 2.16x slowdown
  - Complete checking
  - Highly flexible



- Coming soon: **Intel MPX** hardware
  - Hardware support to make checking faster

https://software.intel.com/en-us/blogs/2013/07/22/intel-memory-protection-extensions-intel-mpx-support-in-the-gnu-toolchain