Memory safety, continued

With material from Mike Hicks, Dave Levin and Michelle Mazurek

http://images.myhardhatstickers.com/img/lg/H/Everybody-Safety-Hard-Hat-Label-HH-0115.gif

Today’s agenda

- Other memory exploits
- Programming Language-level approach to achieving memory safety

Other memory exploits

Heap overflow

- Stack smashing overflows a stack-allocated buffer
- You can also **overflow a buffer** allocated by `malloc`, which resides on the **heap**
  - What data gets overwritten?

Heap overflow

```
typedef struct _vulnerable_struct {
  char buff[MAX_LEN];
  int (*cmp)(char*,char*);
} vulnerable;

int foo(vulnerable* s, char* one, char* two)
{
 strcpy( s->buff, one );
 strcat( s->buff, two );
 return s->cmp( s->buff, "file://foobar" );
}
```

copy `one` into `buff`
copy `two` into `buff`

must have `strlen(one)+strlen(two) < MAX_LEN`

or we overwrite `s->cmp`

Heap overflow variants

- **Overflow into adjacent objects**
  - Where buff is not co-located with a function pointer, but is allocated near one on the heap
- **Overflow heap metadata**
  - Hidden header just before the pointer returned by malloc
  - Flow into that header to corrupt the heap itself

Integer overflow

```
void vulnerable()
{
  char *response;
  int nresp = packet_get_int();
    if (nresp > 0) {
     response = malloc(nresp*sizeof(char*));
     for (i = 0; i < nresp; i++)
      response[i] = packet_get_string(NULL);
    }
```

- What if we set `nresp` =1073741824?
  - Assume `sizeof(char*)` = 4
  - How many bytes are `malloc`’d?
- The `for` loop now creates an overflow!

Stale memory

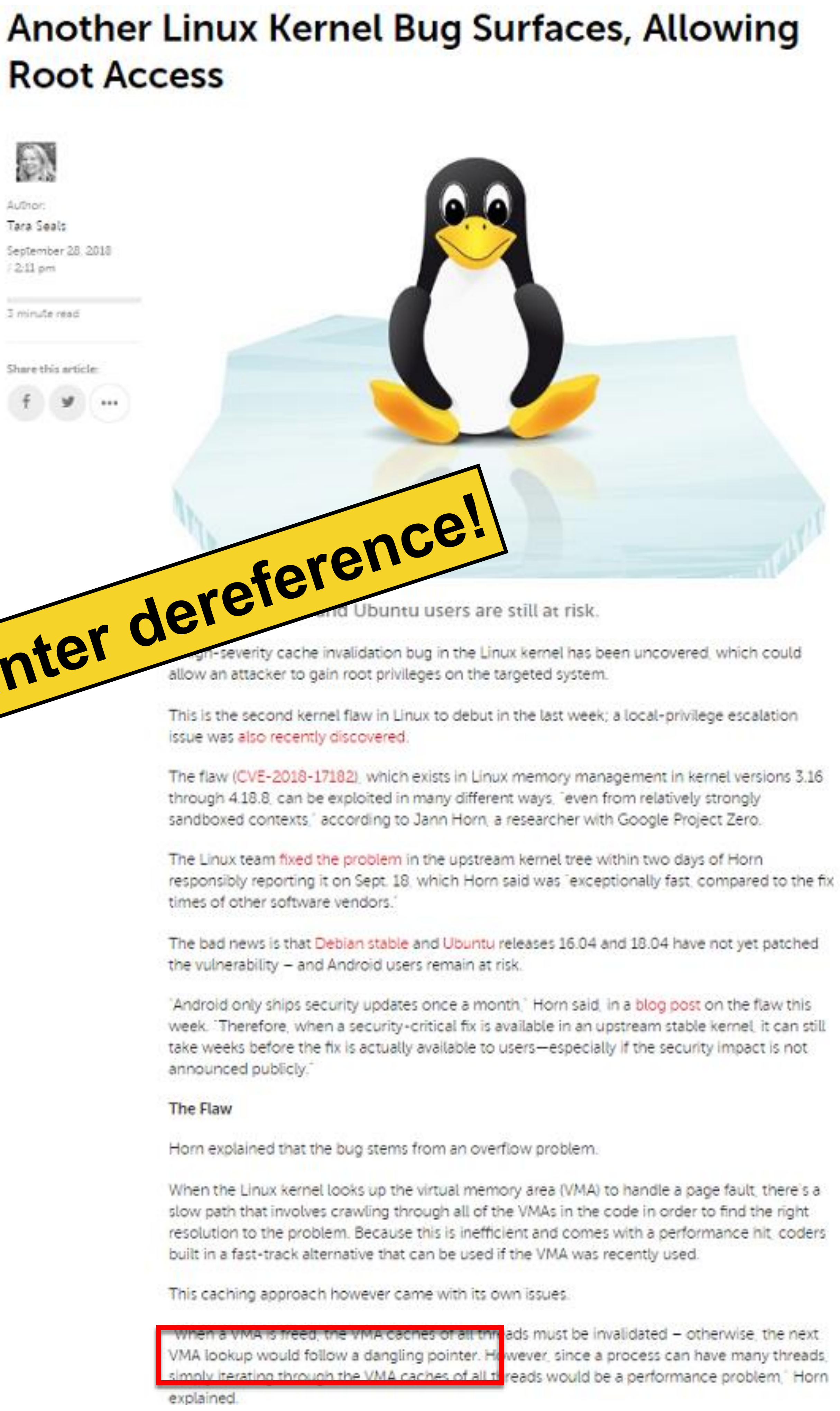
- A **dangling pointer bug** occurs when a pointer is freed, but the program continues to use it
- An attacker can **arrange for the freed memory to be reallocated** and under his control
  - When the dangling pointer is dereferenced, it will access attacker-controlled data

Stale memory

```
struct foo { int (*cmp)(char*,char*); };

struct foo *p = malloc(…);
free(p);
. . .
q = malloc(…) //reuses memory
*q = 0xdeadbeef; //attacker control
. . .
p->cmp(“hello”,”hello”); //dangling ptr
```

- When the dangling pointer is dereferenced, it will access at



Another Linux Kernel Bug Surfaces, Allowing Root Access

Author:
Tara Seals
September 28, 2018 / 2:11 pm

3 minute read

Share this article:

Dangling pointer dereference!

nd Ubuntu users are still at risk.

gh-severity cache invalidation bug in the Linux kernel has been uncovered, which could allow an attacker to gain root privileges on the targeted system.

This is the second kernel flaw in Linux to debut in the last week; a local-privilege escalation issue was also recently discovered.

The flaw (CVE-2018-17182), which exists in Linux memory management in kernel versions 3.16 through 4.18.8, can be exploited in many different ways, "even from relatively strongly sandboxed contexts," according to Jann Horn, a researcher with Google Project Zero.

The Linux team fixed the problem in the upstream kernel tree within two days of Horn responsibly reporting it on Sept. 18, which Horn said was "exceptionally fast, compared to the fix times of other software vendors."

The bad news is that Debian stable and Ubuntu releases 16.04 and 18.04 have not yet patched the vulnerability – and Android users remain at risk.

"Android only ships security updates once a month," Horn said, in a blog post on the flaw this week. "Therefore, when a security-critical fix is available in an upstream stable kernel, it can still take weeks before the fix is actually available to users—especially if the security impact is not announced publicly."

The Flaw

Horn explained that the bug stems from an overflow problem.

When the Linux kernel looks up the virtual memory area (VMA) to handle a page fault, there's a slow path that involves crawling through all of the VMAs in the code in order to find the right resolution to the problem. Because this is inefficient and comes with a performance hit, coders built in a fast-track alternative that can be used if the VMA was recently used.

This caching approach however came with its own issues.

"When a VMA is freed, the VMA caches of all threads must be invalidated – otherwise, the next VMA lookup would follow a dangling pointer. However, since a process can have many threads, simply iterating through the VMA caches of all threads would be a performance problem," Horn explained.

Format string vulnerabilities

Formatted I/O

- Recall: C’s `printf` family of functions
- Format specifiers, list of arguments
  - Specifier indicates type of argument (%s, %i, etc.)
  - Position in string indicates argument to print

```
void print_record(int age, char *name)
{
  printf("Name: %s\tAge: %d\n",name,age);
}
```

What’s the difference?

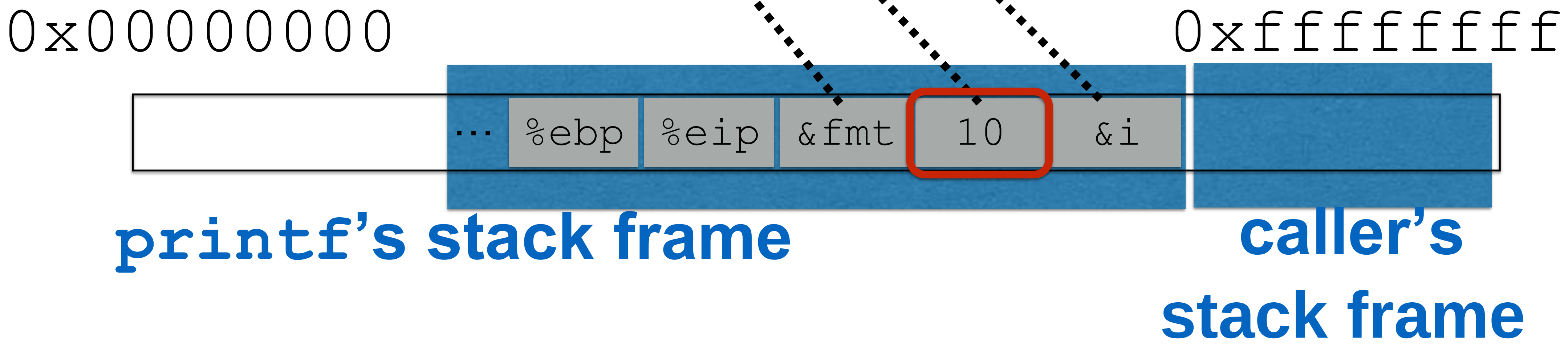
```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

Attacker controls the format string

```
void safe()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf("%s",buf);
}
```

`printf` implementation

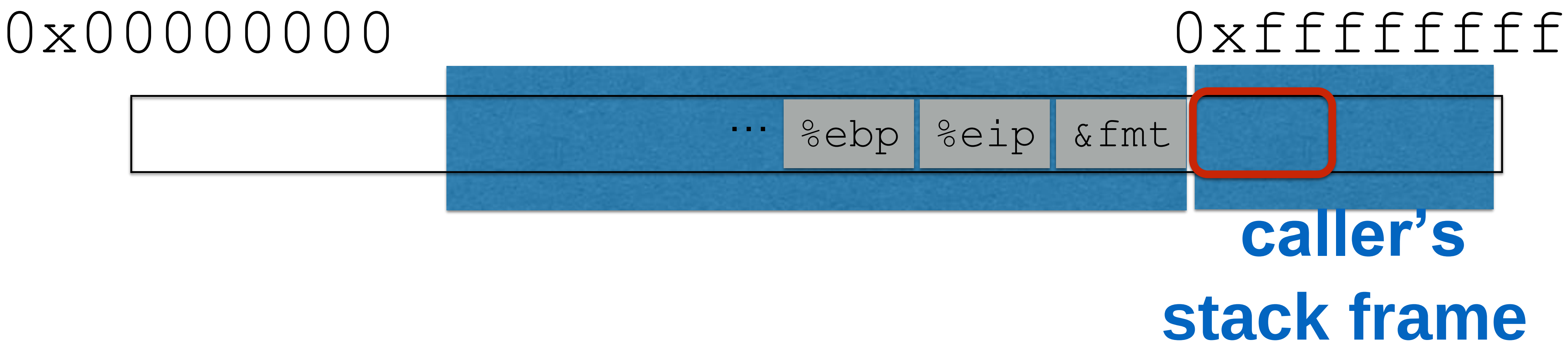
```
int i = 10;
printf("%d %p\n", i, &i);
```



- printf takes a variable number of arguments
- Doesn't know where the stack frame "ends"
- Keeps reading from stack until out of format specifiers

```c
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

" %d %x"



Format string vulnerabilities

- `printf(" 100% dinosaur ");`
  - Prints stack entry 4 byes above saved %eip
- `printf(" %s ");`
  - Prints bytes *pointed to* by that stack entry
- `printf(" %d %d %d %d ..");`
  - Prints a series of stack entries as integers
- `printf(" %08x %08x %08x %08x ..");`
  - Same, but nicely formatted hex
- `printf(" 100% not vulnerable! ")`
  - ***WRITES*** the number 3 to address pointed to by stack entry

Why is this a buffer overflow?

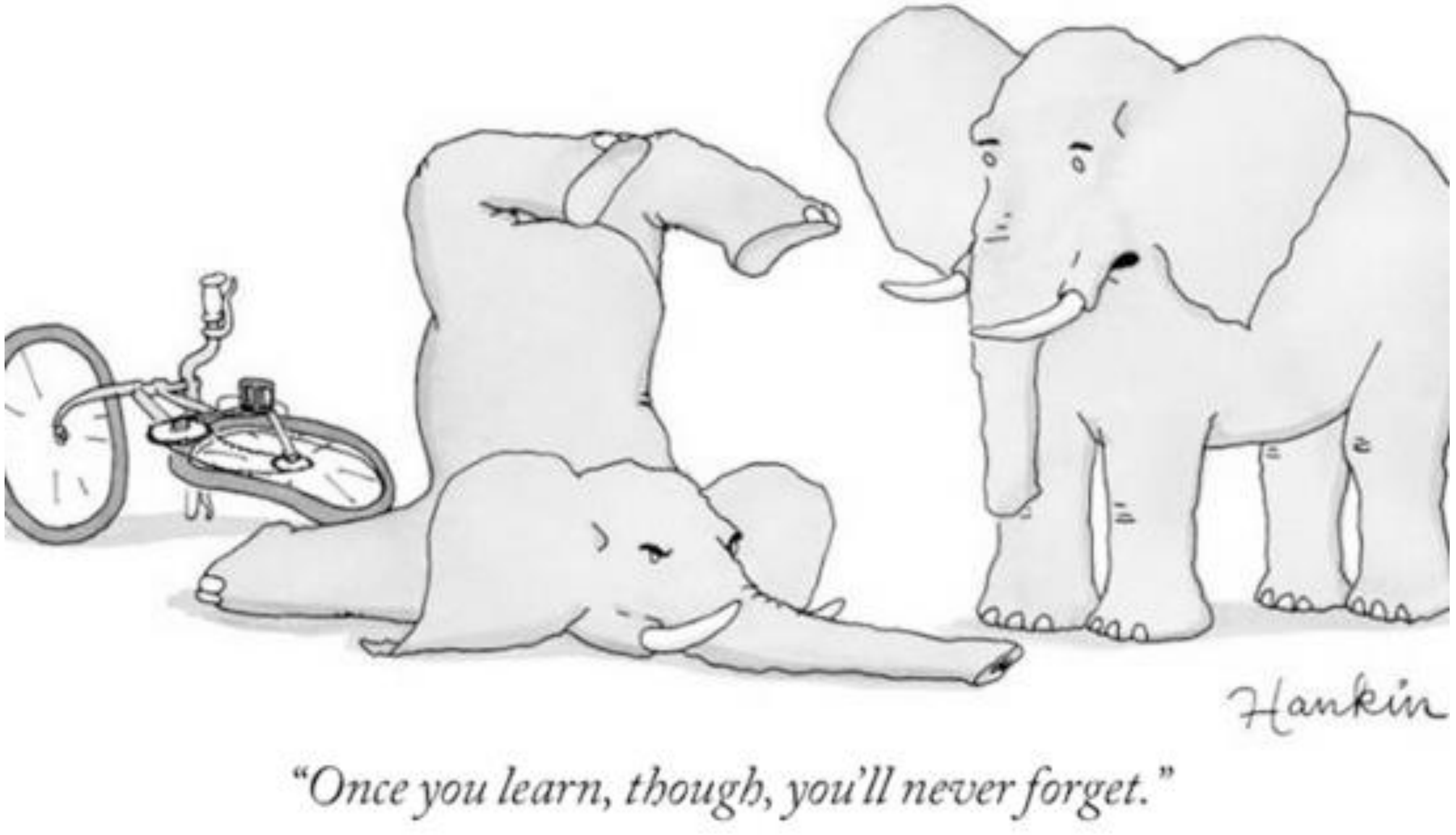
- We should think of this as a buffer overflow in the sense that
  - The stack itself can be viewed as a kind of buffer
  - Size of that buffer is determined by the number and size of the arguments passed to a function
- Providing a bogus format string thus induces the program to overflow that “buffer”

Stepping back

What do these attacks have in common?

1. The attacker is able to **control some data** that is used by the program

2. The use of that data permits **unintentional access to some memory area** in the program
   - Past a buffer
   - To arbitrary positions on the stack / in the heap



Memory Safety

https://diarrheapolice.files.wordpress.com/2015/08/elephant-bicycle-never-forget.png?w=590&h=334

The Basics

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 ***capabilities***: triples (**p**,***b***,***e***)
  - **p** is the actual pointer (current address)
  - ***b*** is the base of the memory region it may access
  - ***e*** is the extent (bounds) of that region (count)
- **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

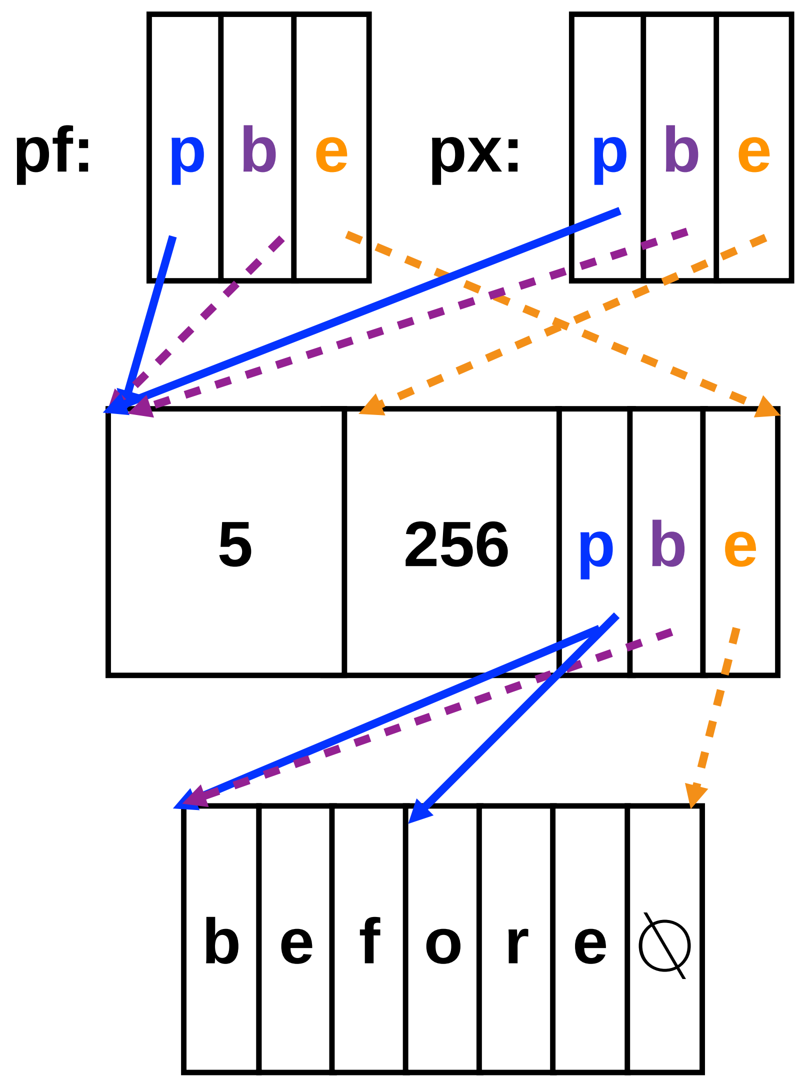
```
int x;          // assume sizeof(int)=4
int *y = &x;    // p = &x, b = &x, e = &x+4
int *z = y+1;   // p = &x+4, b = &x, e = &x+4
*y = 3;         // OK: &x ≤ &x ≤ (&x+4)-4
*z = 3;         // Bad: &x ≤ &x+4 ≰ (&x+4)-4
```

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

```
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 bounds of 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

- Violated 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, no reuse

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?

```
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 are themselves allowed
  - But can’t become illegal pointers
- Integer overflows often enable buffer overflows

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

How to get memory safety?

- 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)

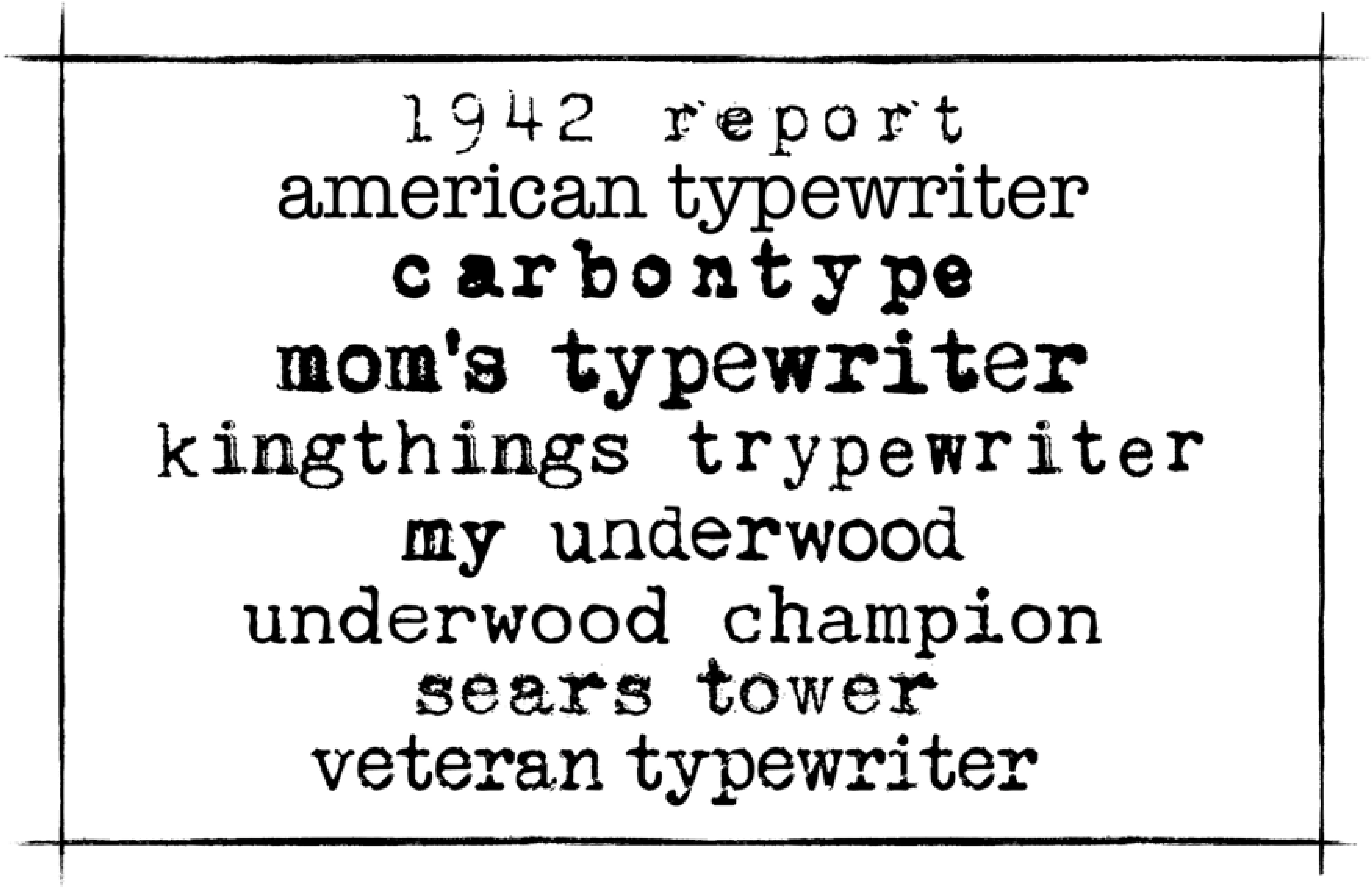
Memory safety for C

- **C/C++ are here to stay**.
  - You **can** write memory safe programs with them
  - But the language provides no guarantee
- Compilers could add code to **check for violations**
  - Out-of-bounds: immediate failure (Java *ArrayBoundsException)*
- 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

Research progress

- **CCured** (2004), 1.5x slowdown
  - But no checking in libraries
  - Compiler rejects many safe programs
- **Softbound/CETS** (2010): 2.16x slowdown
  - Complete checking, highly flexible
- **Intel MPX** hardware (2015 in Linux)
  - 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



Type Safety

https://a.dilcdn.com/bl/wp-content/uploads/sites/8/2014/02/typewriterfont.jpg

Type safety

- Each object is ascribed a **type** (`int`, pointer to `int`, pointer to function), and
- Operations on the object are always *compatible* with the object's type
  - Type safe programs do not "go wrong" at run-time
- **Type safety** is **stronger** than memory safety

```
int (*cmp)(char*,char*);
int *p = (int*)malloc(sizeof(int));
*p = 1;
cmp = (int (*)(char*,char*))p;
cmp("hello","bye"); // crash!
```

Memory safe,
NOT type safe

Aside: Dynamic Typing

- Dynamically typed languages
  - Don’t require type declaration
  - e.g., Ruby and Python
  - Can be viewed as type safe
- Each object has **one type: Dynamic**
  - Each operation on a Dynamic object is permitted, but *may be unimplemented*
  - In this case, it *throws an exception*
  - Checked at **runtime** not **compile time!**

Types for Security

- Use types to enforce **security property** invariants
  - Invariants about data's privacy and integrity
  - Enforced by the type checker
- **Example**: **Java with Information Flow (JIF)**

```
int{Alice, Bob} x;
int{Alice, Bob, Chuck} y;
x = y; //OK: policy on x is stronger
y = x; //BAD: policy on y is weaker
```

Types have ***security labels*** that govern ***information flow***

http://www.cs.cornell.edu/jif

Why not type safety?

- C/C++ often chosen **for performance** reasons
  - Manual memory management
  - Tight control over object layouts
  - Interaction with low-level hardware
- **Enforcement** of type safety is typically **expensive**
  - **Garbage collection** avoids temporal violations
    - Can be as fast as malloc/free, often uses much more memory
  - **Bounds** and **null-pointer checks** avoid spatial violations
  - **Hiding representation** may **inhibit optimization**
    - Many C-style casts, pointer arithmetic, & operator, not allowed

A new hope?

- Many applications do not need C/C++
  - Or the risks that come with it
- New languages aim to provide **similar features** to C/C++ while **remaining type safe**
  - Google’s **Go**, Mozilla’s **Rust**, Apple’s **Swift**