so far

many vulnerabilities we looked at due to poor bounds checking

- one exception: use-after-free and related

can we just fix this?

adding bounds checking

```
char buffer[42];
memcpy(buffer, attacker_controlled, len);
```

couldn't compiler add check for `len`

modern Linux: it does

added bounds checking

```
char buffer[42];
memcpy(buffer, attacker_controlled, len);
```

```
    subq   $72, %rsp
    leaq   4(%rsp), %rdi
    movslq len, %rdx
    movq   attacker_controlled, %rsi
    movl   $42, %ecx
    call   __memcpy_chk
```

length 42 passed to `__memcpy_chk`

_FORTIFY_SOURCE

Linux C standard library + GCC features

adds automatic checking to a bunch of string/array functions

also printf (disable %n unless format string is a constant)

often enabled by default

GCC options:

`-D_FORTIFY_SOURCE=1` — enable (backwards-compatible only)
`-D_FORTIFY_SOURCE=2` — enable (full)
`-U_FORTIFY_SOURCE` — disable

bounds checking will happen...

will add checks (gcc 9.3 -**O2**)

```
void example1() {
    char dest1[1024]; memcpy(dest1, ...); ...
}
char dest2[1024];
void example2() {
    memcpy(dest2, ...); ...
}
void example3() {
    char *p = &dest2[4]; memcpy(p, ...); ...
}
```

bounds checking won't happen...

will not add check (gcc 9.3 -**O2**)

```
char dest2[1024];
void example4() {
    char *p = &dest2[mystery()]; memcpy(p, ...); ...
}
```

adds check for size 1024 (max possible size):

```
char dest2[1024];
void example5() {
    char dest3[128];
    char *p = dest2;
    if (mystery()) p = dest3;
    memcpy(p, ...); ...
}
```

non-checking library functions

some C library functions make bounds checking hard:

```
strcpy(dest, source);
strcat(dest, source);
sprintf(dest, format, ...);
```

bounds-checking versions (*added to library later*):

```
/* might not add \0 (!) */
strncpy(dest, source, size);
strncat(dest, source, size);
snprintf(dest, size, format, ...);
```

poor bounds-checking APIs

```
char dest[100];
/* THIS CODE IS BROKEN */
strncpy(dest, source1, sizeof dest);
strncat(dest, source2, sizeof dest);
printf("result␣was␣%s\n", dest)
```

the above can access memory of out of bounds

...in a bunch of ways

Linux's strncpy manual

```
strncpy(dest, source1, sizeof dest);
```

"Warning: If there is no null byte among the first n bytes of src, the string placed in dest will not be null-terminated."

exercise: what should the call have been?

Linux's strncat manual

```
strncat(dest, source2, sizeof dest);
```

"If src contains n or more bytes, strncat() writes $n+1$ bytes to dest (n from src plus the terminating null byte). Therefore, the size of dest must be at least $\text{strlen(dest)}+n+1$."

exercise: what should the call have been?

better versions?

FreeBSD (and Linux via libbsd): strlcpy, strlcat

“Unlike [strncat and strncpy], strlcpy() and strlcat() take the full size of the buffer and gaurenteeto NUL-terminate the result...”

```
strlcpy(dest, source1, sizeof dest);
strlcat(dest, source2, sizeof dest);
```

Windows: `strcpy_s`, `strcat_s` (same idea, differentname)

C++ bounds checking

```cpp
#include <vector>
...
std::vector<int> data;
data.resize(50);
// undefined behavior:
data[60] = 0;
// throws std::out_of_range exception
data.at(60) = 0;
```

language-level solutions

languages like Python don't have this problem

couldn't we do the same thing in C?

bounds-checking C

there have been many proposals to add bounds-checking to C

including implementations

brainstorm: *why hasn't this happened?*

easy bounds-checking

```
void vulnerable() {
    char buffer[100];
    int c;
    int i = 0;
    while ((c = getchar()) != EOF && c != '\n') {
        buffer[i] = c;
    }
}
void vulnerable_checked() {
    char buffer[100];
    int c;
    int i = 0;
    while ((c = getchar()) != EOF && c != '\n') {
        FAIL_IF(i >= 100 || i < 0);
        buffer[i] = c;
    }
}
```

harder bounds-checking

```
void vulnerable(char *buffer) {
    char buffer[100];
    int c;
    int i = 0;
    while ((c = getchar()) != EOF && c != '\n') {
        buffer[i] = c;
    }
}
void vulnerable_checked(char *buffer) {
    int c;
    int i = 0;
    while ((c = getchar()) != EOF && c != '\n') {
        FAIL_IF(i >= UNKNOWN || i < UNKNOWN);
        buffer[i] = c;
    }
}
```

adding bounds-checking — fat pointers

```
struct MyPtr {
    char *pointer; /* "raw" pointer value */
    char *minimum; /* first byte of buffer pointed to */
    char *maximum; /* last byte of buffer pointed to */
};
```

adding bounds-checking — fat pointers

```
struct MyPtr {
    char *pointer; /* "raw" pointer value */
    char *minimum; /* first byte of buffer pointed to */
    char *maximum; /* last byte of buffer pointed to */
};
```

```
char buffer[100];
char *p = &buffer[10];
```

becomes

```
char buffer[100];
MyPtr p = {
    .pointer = &buffer[10],
    .minimum = &buffer[0],
    .maximum = &buffer[99]
};
```

adding bounds checking — strcpy

```
MyPtr strcpy(MyPtr dest, const MyPtr src) {
    int i;
    do {
        CHECK(src.pointer + i <= src.maximum);
        CHECK(src.pointer + i >= src.minimum);
        CHECK(dest.pointer + i <= dest.maximum);
        CHECK(dest.pointer + i >= dest.minimum);
        dest.pointer[i] = src.pointer[i];
        i += 1;
        CHECK(src.pointer + i <= src.maximum);
        CHECK(src.pointer + i >= src.minimum);
    } while (src.pointer[i] != '\0');
    return dest;
}
```

speed of bounds checking

two comparisons for every pointer access?

three times as much space for every pointer?

unfortunate things C programmers do (1)

from FreeBSD's bootpd (server for machines that boot from the network):

```
struct shared_string {
    unsigned int linkcount;
    char         string[1]; /* Dynamically extended */
};
...
s = (struct shared_string *) smalloc(
        sizeof(struct shared_string) + length
    );
...
```

unfortunate things C programmers do (2)

from perl's source code:

```
sv_setuv(my_pool_sv, PTR2UV(my_poolp));
...
/* later, in another function: */
my_pool_t *my_poolp = INT2PTR(my_pool_t*, SvUV(my_pool_sv));
```

PTR2UV: pointer to Unsigned int Value

INT2PTR: integer to pointer value

unfortunate things C programmers do (3)

```
struct SuperClass;
struct SubClass {
    struct SuperClass super;
    ...
}

struct SubClass sub;
struct SuperClass *super = &sub.super;
some_function(super);
...
some_function(struct SuperClass *super) {
    ...
    struct SubClass *sub = (struct SubClass *)super;
    ...
}
```

example: CCured

Necula et al, "CCured: Type-Safe Retrofitting of Legacy Code" (2002)

extension to C to add fat pointers

actually three different types of pointers:

- SAFE: point to single object (not array) or NULL
- SEQUENCE: pointer to array with known bounds (like "fat" pointers)
- DYNAMIC: extra to handles type-casting

needs source changes to annotate some pointer usage

- especially to allow library function calls

1-**2.5x** time overhead

research example (2009)

Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors

baggy bounds checking idea

giant lookup table — one entry for every 16 bytes of memory

table indicates start of object allocated here

check pointer arithmetic:

```
char p = str[i];
/* becomes: */
CHECK(START_OF[str / 16] == START_OF[&str[i] / 16])
char p = str[i];
```

baggy bounds trick

table of pointers to starting locations would be huge

add some restrictions:

- all object sizes are powers of two
- all object starting addresses are a multiple of their size

then, table contains size info only:

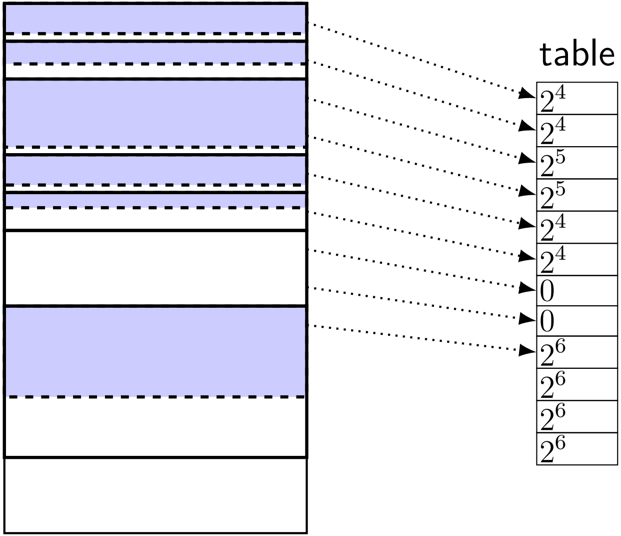
- table contains i, size is 2^i bytes:

```
char *GetStartOfObject(char *pointer) {
    return pointer & ~(1 << TABLE[pointer / 16] - 1);
    /* pointer bitwise-and 2^(table entry) - 1 */
    /* clear lower (table entry) bits  of pointer */
}
```

allocations and lookup table

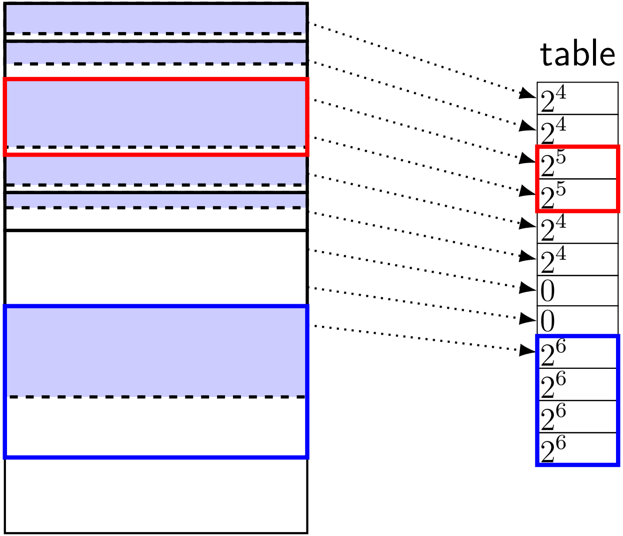
object allocated in *power-of-two 'slots'*

allocations and lookup table



object allocated in power-of-two 'slots'

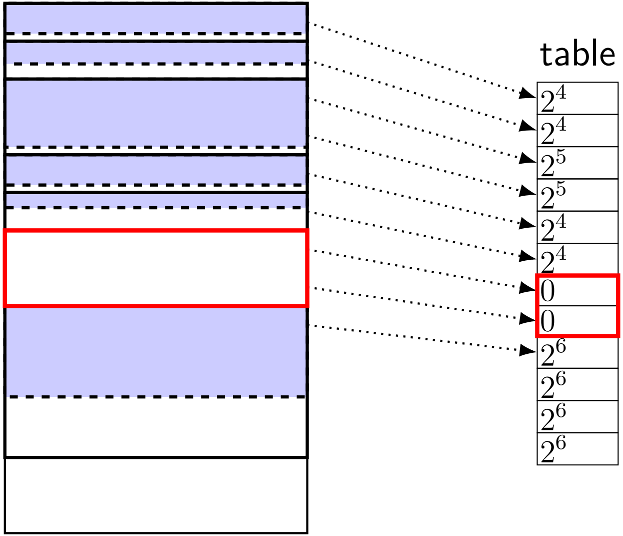
allocations and lookup table



object allocated in power-of-two 'slots'

table stores sizes *for each 16 bytes*

allocations and lookup table

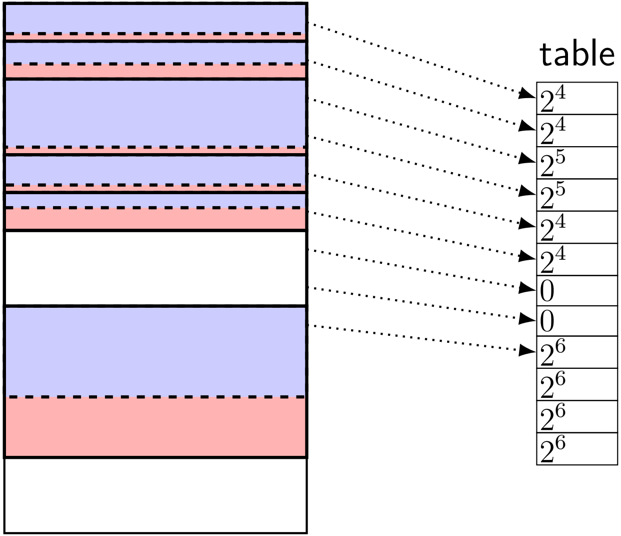


object allocated in power-of-two 'slots'

table stores sizes *for each 16 bytes*

addresses **multiples of size** (may *require padding*)

allocations and lookup table



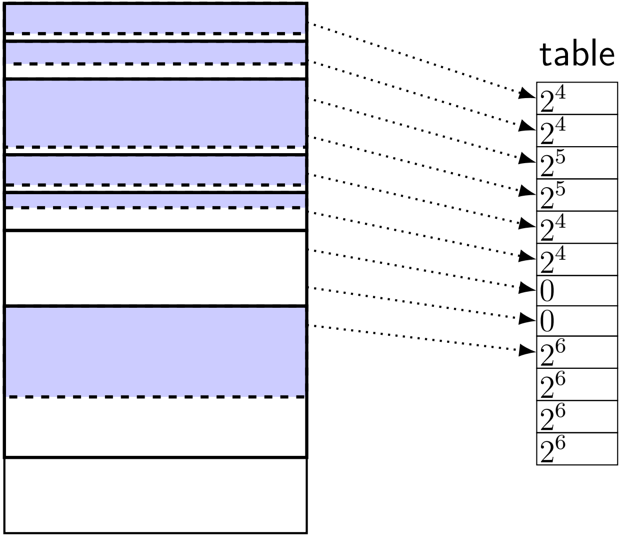
object allocated in power-of-two 'slots'

table stores sizes *for each 16 bytes*

addresses multiples of size (may *require padding*)

sizes are **powers of two** (may *require padding*)

allocations and lookup table



object allocated in power-of-two 'slots'

table stores sizes *for each 16 bytes*

addresses multiples of size (may *require padding*)

sizes are powers of two (may *require padding*)

managing the table

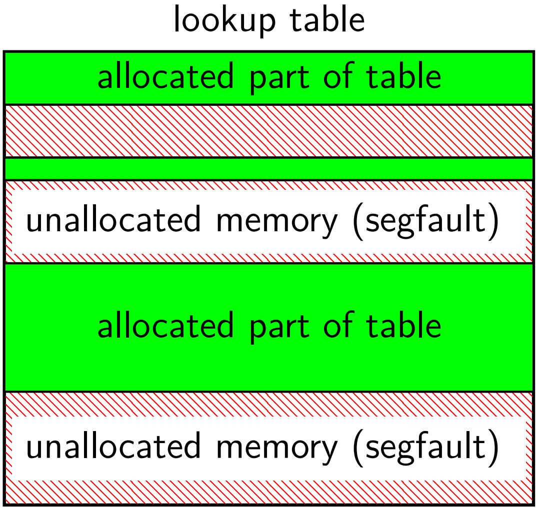
not just done `malloc()/new`

also for stack allocations:

```
void vulnerable() {
    char buffer[100];
    gets(vulnerable);
}
```

```
vulnerable:
    // make %rsp a multiple
    // of 128 (2^7)
    andq $0xFFFFFFFFFFFFFF80, %rsp
    // allocate 128 bytes
    subq $0x80, %rsp
    // rax <- rsp / 16
    movq $rsp, %rax
    shrq $4, %rax
    movb $7, TABLE(%rax)
    movb $7, TABLE+1(%rax)
    ...
    movq %rsp, %rdi
    call gets
    ret
```

sparse lookup table



baggy bounds check: added code

bounds lookup

```
mov eax, buf
shr eax, 4
mov al, byte ptr [TABLE+eax]
```

pointer arithmetic

```
char *p = buf[i];
```

bounds check

```
mov ebx, buf
xor ebx, p
shr ebx, al
jz ok
  p = slowPath(buf, p)
ok:
```

Figure 5: Code sequence inserted to check unsafe pointer arithmetic.

baggy bounds check: added code

```
/* bounds lookup */
    mov buf, %rax
    shr %rax, 4
    mov LOOKUP_TABLE(%rax), %al
/* array element address computation */
    ...    // char * p = buf[i];
/* bound check */
    mov buf, %rbx
    xor p, %rbx
    shr %al, %rbx
    jz  ok
    ...    // handle possible violation
ok:
```

adapted from paper figure

avoiding checks

code not added if not array/pointer accesses to object

code not added when pointer accesses “obviously” safe

- author’s implementation: only checked within function

exercise: overhead of baggy bounds (1)

suppose program allocates:

- 1000 100 byte objects
- 1 10000 byte object

using baggy bounds, estimate:

- space required for padding

- space required for table

exercise: overhead of baggy bounds (1)

suppose program allocates:

- 1000 100 byte objects
- 1 10000 byte object

using baggy bounds, estimate:

- space required for padding
  - $(128 - 100) \cdot 1000 + (16384 - 10000)) = 34384$
- space required for table
  - $(128 \cdot 1000 + 16384) \div 16 = 9024$

exercise: overhead of baggy bounds (2)

```
char *strcat(char *d, char *s) {
    int i;
    for (i = 0; s[i] != '\0'; i += 1) {
        d[i] = s[i];
    }
    d[i] = '\0';
    return d;
}
```

estimate:

number of bounds checks needed

very rough number of instructions run w/o bounds check

thought question:

with bounds checking, what's fastest possible code?

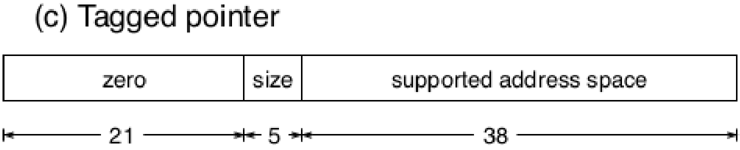
alternate approach: pointer tagging

some bits of *address* are size

replaces table entry/lookup

change code to allocate objects this way

works well on 64-bit — plenty of addresses to use



baggy bounds performance

table: 4–72% time overhead (depends on benchmark suite)

table: 11–21% space overhead (depends on benchmark suite)

tagged pointers: slightly better on average

baggy bounds performance

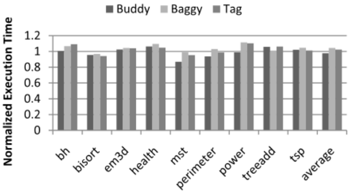


Figure 19: Normalized execution time on AMD64 with Olden benchmarks.

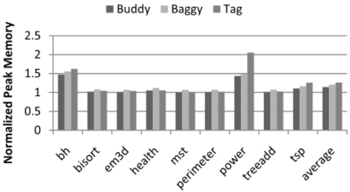


Figure 21: Normalized peak memory use on AMD64 with Olden benchmarks.

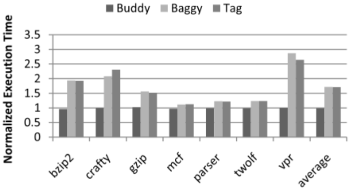


Figure 20: Normalized execution time on AMD64 with SPECINT 2000 benchmarks.

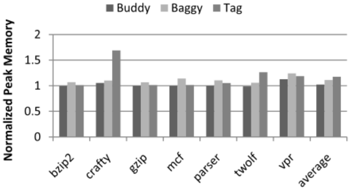


Figure 22: Normalized peak memory use on AMD64 with SPECINT 2000 benchmarks.

problem: within object

```
struct foo {
    char buffer[1024];
    int *pointer;
};
struct foo array_of_foos[1024];
...
char *p = &array_of_foos[4].buffer[4]
```

exercise: what are the bounds for p?

unfortunate things C programmers do (4)

in code generated by f2c (Fortran to C translator)

(cleaned up slightly)

```
float sum(int size, float *arr) {
    arr = arr - 1; /* <-- deliberately out-of-bounds pointer *
    float result = 0.f;
    for (i = 1; i <= size; ++i) {
        result += arr[i]
    }
    return result;
}
```

AddressSanitizer

like baggy bounds:

- big lookup table
- lookup table set by memory allocations
- compiler modification: change stack allocations

unlike baggy bounds:

- check reads/writes (instead of pointer computations)
- only detect errors that read/write *between objects*
- object sizes not padded to power of two
- table has info for every single byte (more precise)

adding bounds-checking example

```
void vulnerable(long value, int offset) {
    long array[10] = {1,2,3,4,5,6,7,8,9,10};
    // generated code: (added by AddressSanitizer)
    if (!lookup_table[&array[offset]] == VALID) FAIL();
    array[offset] = value;
    do_something_with(array);
}
```

AddressSanitizer: crashes only if `array[offset]` isn't part of any object

- but no extra space — single-byte precision

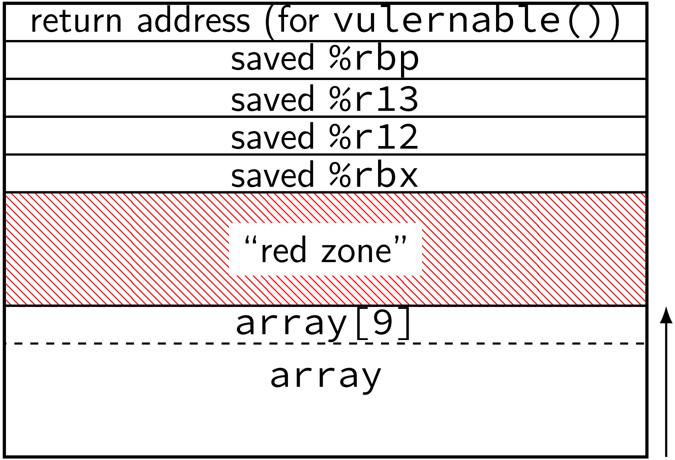
adding bounds-checking example

```
void vulnerable(long value, int offset) {
    long array[10] = {1,2,3,4,5,6,7,8,9,10};
    // generated code: (added by AddressSanitizer)
    if (!lookup_table[&array[offset]] == VALID) FAIL();
    array[offset] = value;
    do_something_with(array);
}
```

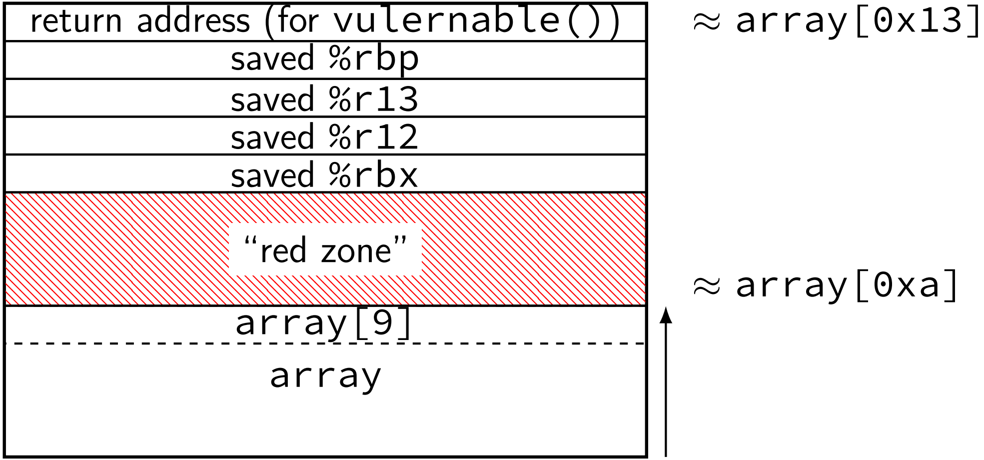
AddressSanitizer: crashes only if `array[offset]` isn't part of any object

but no extra space — single-byte precision

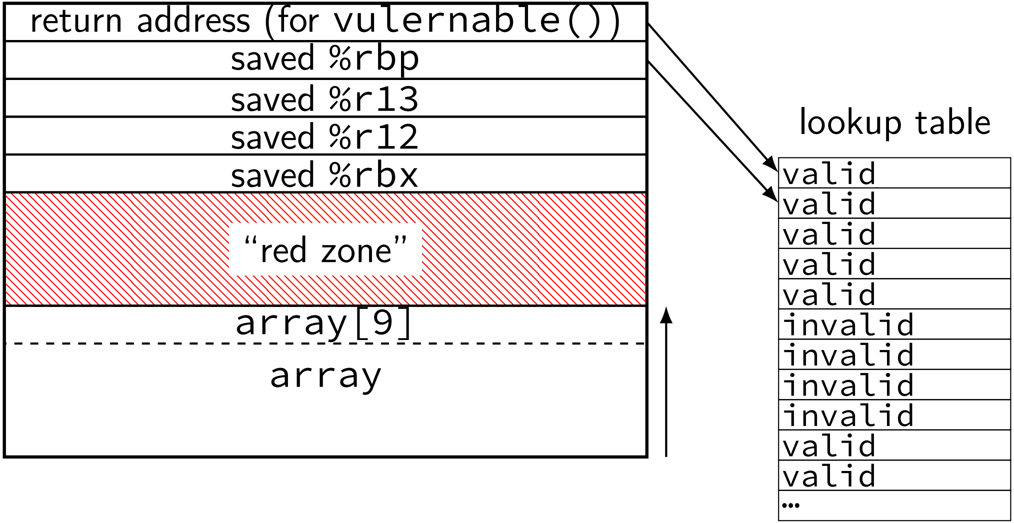
AddressSanitizer stack layout



AddressSanitizer stack layout



AddressSanitizer stack layout



AddressSanitizer

like baggy bounds:

- big lookup table
- lookup table set by memory allocations
- compiler modification: change stack allocations

unlike baggy bounds:

- check reads/writes (instead of pointer computations)
- only detect errors that read/write *between objects*
- object sizes not padded to power of two
- table has info for every single byte (more precise)

adding bounds-checking example

```
void vulnerable(long value, int offset) {
    long array[10] = {1,2,3,4,5,6,7,8,9,10};
    // generated code: (added by AddressSanitizer)
    if (!lookup_table[&array[offset]] == VALID) FAIL();
    array[offset] = value;
    do_something_with(array);
}
```

AddressSanitizer: crashes only if `array[offset]` isn't part of any object

- but no extra space — single-byte precision

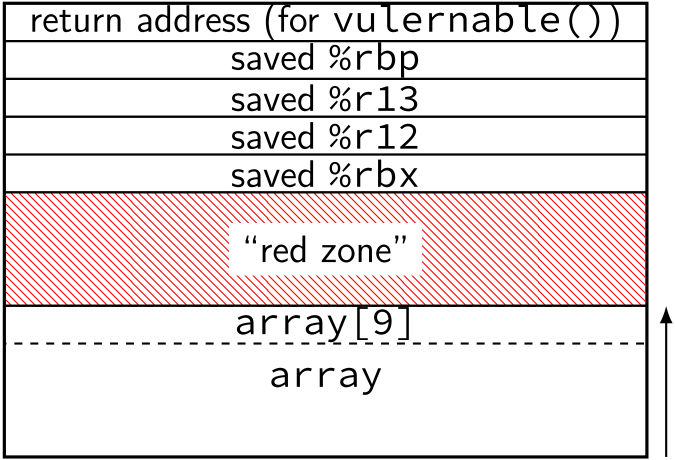
adding bounds-checking example

```
void vulnerable(long value, int offset) {
    long array[10] = {1,2,3,4,5,6,7,8,9,10};
    // generated code: (added by AddressSanitizer)
    if (!lookup_table[&array[offset]] == VALID) FAIL();
    array[offset] = value;
    do_something_with(array);
}
```

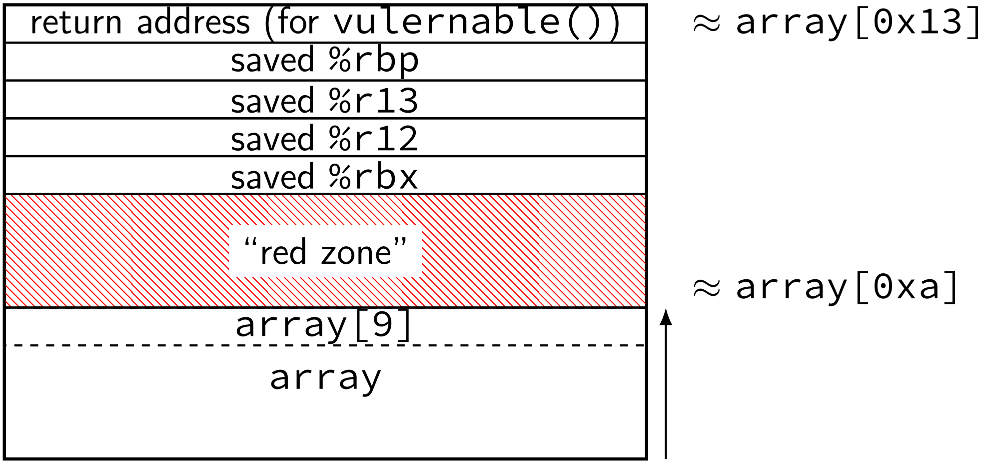
AddressSanitizer: crashes only if `array[offset]` isn't part of any object

but no extra space — single-byte precision

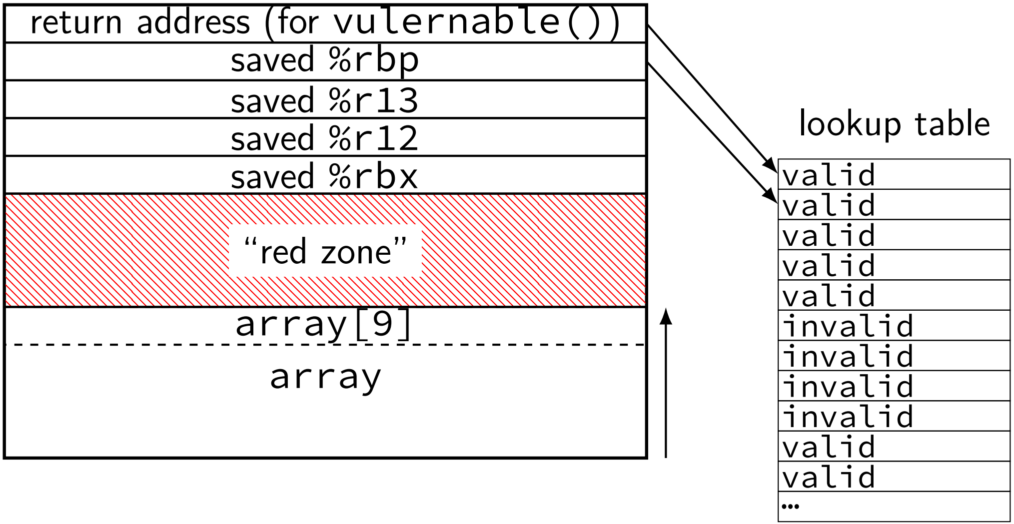
AddressSanitizer stack layout



AddressSanitizer stack layout

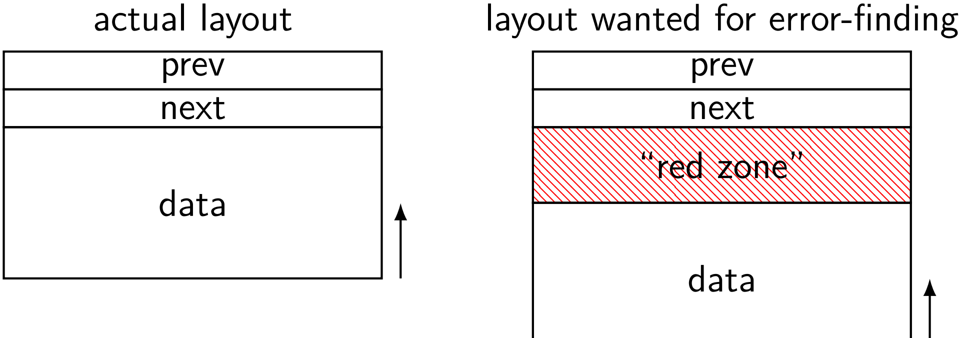


AddressSanitizer stack layout



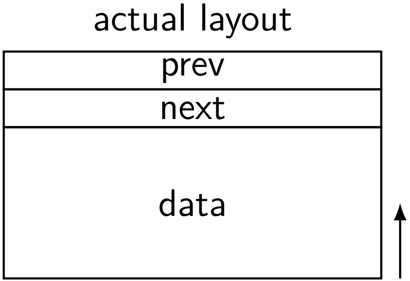
changing object layout?

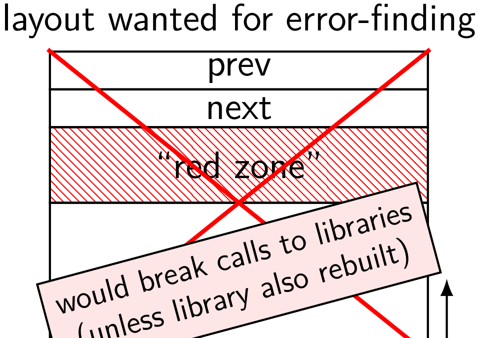
```
struct string_list {
    char data[100];
    struct string_list *prev;
    struct string_list *next;
};
```



changing object layout?

```
struct string_list {
    char data[100];
    struct string_list *prev;
    struct string_list *next;
};
```





AddressSanitizer versus Baggy Bounds

pros vs baggy bounds:

- you can actually use it (comes with GCC/Clang)
- byte-level precision — no "padding" on objects
- detects use-after-free a lot of the time

cons vs baggy bounds:

- doesn't prevent out-of-bounds "targetted" accesses
- requires extra space between objects
- usually slower

Valgrind Memcheck

similar to AddressSanitizer — but no compiler modificaitons

instead: is a virtual machine (plus alternate malloc/new implementation)

only (reliably) detects errors on heap

but works on *unmodified* binaries

which scheme prevents...?

which schemes detect or prevent from being harmful...?

1. call to assembly code that goes beyond buffer?
2. allowing attacker to insert 150 bytes in 100 byte buffer on heap?
3. allowing attacker to insert 120 bytes in 100 byte buffer on stack?
4. attecker exploiting code that does array[attacker_index] to overwrite something outside heap array?

of:

A. "fat pointers" approach
B. Baggy Bounds checking
C. AddressSanitizer
D. Valgrind Memcheck