Verification and Testing

Memory Debugging

Vedad Hadžić

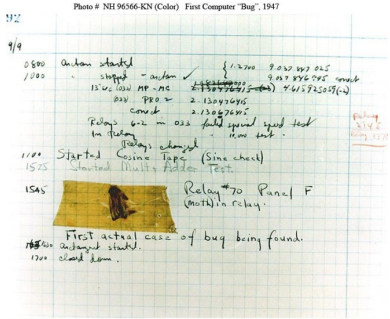
20.10.2022

Winter 2022/23, www.iaik.tugraz.at

Table of contents

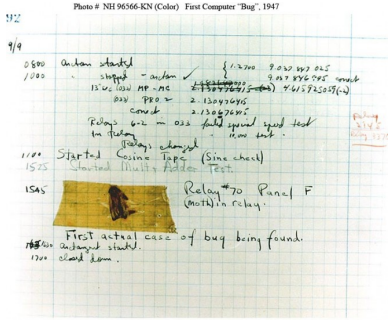
Introduction

- Have you ever debugged a segfault?



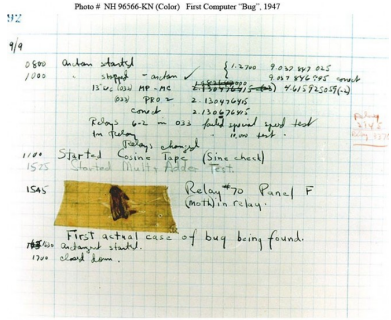
Debugging Code

- Have you ever debugged a segfault?
  1. Program crashes unexpectedly
  2. You investigate the crash site
  3. Trace it back, fix it



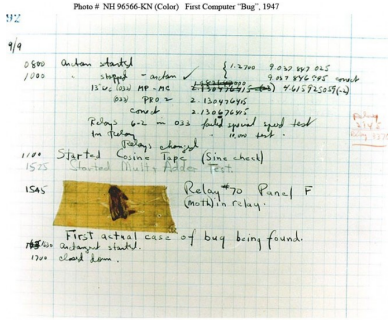
Debugging Code

- Have you ever debugged a segfault?
- What about memory leaks?



Debugging Code

- Have you ever debugged a segfault?
- What about memory leaks?
  1. You get told to fix memory leaks (usually)
  2. You run your program through, e.g., Valgrind
  3. After a very long coffee break, you get a report
  4. Trace it back, fix it



Debugging Code

- Have you ever debugged a segfault?
- What about memory leaks?
- Have you ever felt like your code is haunted?

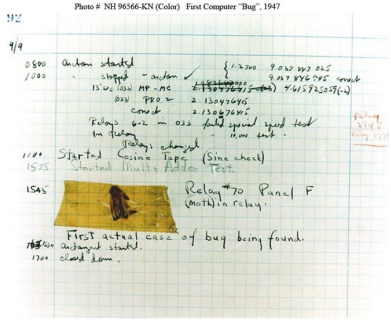
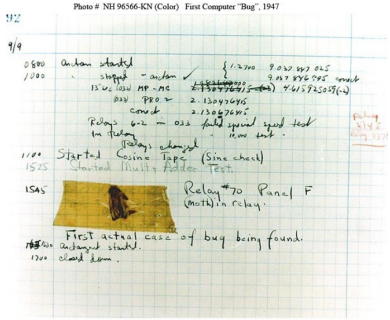


Photo # NH 96566-KN (Color) First Computer "Bug", 1947

Debugging Code

- Have you ever debugged a segfault?
- What about memory leaks?
- Have you ever felt like your code is haunted?
  1. You notice your results are wrong
  2. You cross-check everything a thousand times
  3. You investigate line-by-line with a debugger
  4. You notice the undefined behavior, fix it



- Use after free

```
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc];
}
```

- Use after free

```
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc];// BOOM
}
```

- Use after free

```
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc];// BOOM
}
```

- Heap buffer overflow

```
int main(int argc, char **argv) {
  int *array = new int[100];
  array[0] = 0;
  int res = array[argc + 100];
  delete [] array;
  return res;
}
```

- Use after free

```
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc];// BOOM
}
```

- Heap buffer overflow

```
int main(int argc, char **argv) {
  int *array = new int[100];
  array[0] = 0;
  int res = array[argc + 100];// BOOM
  delete [] array;
  return res;
}
```

- Use after free

```
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc];// BOOM
}
```

- Heap buffer overflow

```
int main(int argc, char **argv) {
  int *array = new int[100];
  array[0] = 0;
  int res = array[argc + 100];// BOOM
  delete [] array;
  return res;
}
```

- Stack buffer overflow

```
int main(int argc, char **argv) {
  int array[100];
  array[1] = 0;
  return array[argc + 100];
}
```

- Use after free

```
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc];// BOOM
}
```

- Heap buffer overflow

```
int main(int argc, char **argv) {
  int *array = new int[100];
  array[0] = 0;
  int res = array[argc + 100];// BOOM
  delete [] array;
  return res;
}
```

- Stack buffer overflow

```
int main(int argc, char **argv) {
  int array[100];
  array[1] = 0;
  return array[argc + 100];// BOOM
}
```

- Use after free

```
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc];// BOOM
}
```

- Heap buffer overflow

```
int main(int argc, char **argv) {
  int *array = new int[100];
  array[0] = 0;
  int res = array[argc + 100];// BOOM
  delete [] array;
  return res;
}
```

- Stack buffer overflow

```
int main(int argc, char **argv) {
  int array[100];
  array[1] = 0;
  return array[argc + 100];// BOOM
}
```

- Global buffer overflow

```
int array[100] = {-1};
int main(int argc, char **argv) {
  return array[argc + 100];
}
```

Memory Errors

- Use after free

```
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc];// BOOM
}
```

- Heap buffer overflow

```
int main(int argc, char **argv) {
  int *array = new int[100];
  array[0] = 0;
  int res = array[argc + 100];// BOOM
  delete [] array;
  return res;
}
```

- Stack buffer overflow

```
int main(int argc, char **argv) {
  int array[100];
  array[1] = 0;
  return array[argc + 100];// BOOM
}
```

- Global buffer overflow

```
int array[100] = {-1};
int main(int argc, char **argv) {
  return array[argc + 100];// BOOM
}
```

- Use after scope/return

```
volatile int *p = 0;
int main() {
  {
    int x = 0;
    p = &x;
  }
  *p = 5;
  return 0;
}
```

Memory Errors Cont.

- Use after scope/return

```
volatile int *p = 0;
int main() {
  {
    int x = 0;
    p = &x; // BUG
  }
  *p = 5; // BOOM
  return 0;
}
```

Memory Errors Cont.

- Use after scope/return

```
volatile int *p = 0;
int main() {
  {
    int x = 0;
    p = &x; // BUG
  }
  *p = 5; // BOOM
  return 0;
}
```

- Memory leaks

```
int main(int argc, char **argv) {
  int *array = new int[100];
  array = 0;
  return 0;
}
```

- Use after scope/return

```
volatile int *p = 0;
int main() {
  {
    int x = 0;
    p = &x; // BUG
  }
  *p = 5; // BOOM
  return 0;
}
```

- Memory leaks

```
int main(int argc, char **argv) {
  int *array = new int[100];
  array = 0;// BOOM
  return 0;
}
```

Memory Errors Cont.

- Use after scope/return

```
volatile int *p = 0;
int main() {
  {
    int x = 0;
    p = &x; // BUG
  }
  *p = 5; // BOOM
  return 0;
}
```

- Memory leaks

```
int main(int argc, char **argv) {
  int *array = new int[100];
  array = 0;// BOOM
  return 0;
}
```

- Uninitialized memory

```
#include <stdio.h>

int main(int argc, char** argv) {
  int* a = new int[10];
  a[5] = 0;
  if (a[argc])
    printf("xx\n");
  return 0;
}
```

- Use after scope/return

```
volatile int *p = 0;
int main() {
  {
    int x = 0;
    p = &x; // BUG
  }
  *p = 5; // BOOM
  return 0;
}
```

- Memory leaks

```
int main(int argc, char **argv) {
  int *array = new int[100];
  array = 0;// BOOM
  return 0;
}
```

- Uninitialized memory

```
#include <stdio.h>

int main(int argc, char** argv) {
  int* a = new int[10];
  a[5] = 0;
  if (a[argc]) // BOOM
    printf("xx\n");
  return 0;
}
```

Does it Segfault?

- Segfaults are good for debugging!

Does it Segfault?

- Segfaults are good for debugging!
- However, they do not always happen.

Does it Segfault?

- Segfaults are good for debugging!
- However, they do not always happen.

- Uninitialized read

```
int a, b;
a = b; // NO SEGFAULT
```

Does it Segfault?

- Segfaults are good for debugging!
- However, they do not always happen.

- Uninitialized read

```
int a, b;
a = b; // NO SEGFAULT
```

- Reading the heap

```
int* a = new int [5];
int b = a[7]; // USUALLY NO SEGFAULT
```

Does it Segfault?

- Segfaults are good for debugging!
- However, they do not always happen.

- Uninitialized read

```
int a, b;
a = b; // NO SEGFAULT
```

- Reading the heap

```
int* a = new int [5];
int b = a[7]; // USUALLY NO SEGFAULT
```

- Writing the heap

```
int* a = new int [5];
a[7] = 14;     // MAYBE NO SEGFAULT
```

Does it Segfault?

- Segfaults are good for debugging!
- However, they do not always happen.

- Uninitialized read

```
int a, b;
a = b; // NO SEGFAULT
```

- Reading the heap

```
int* a = new int [5];
int b = a[7]; // USUALLY NO SEGFAULT
```

- Writing the heap

```
int* a = new int [5];
a[7] = 14;     // MAYBE NO SEGFAULT
```

- Using freed memory

```
int* a = new int [5];
delete [] a;
int b = a[3]; // USUALLY NO SEGFAULT
```

Does it Segfault?

- Segfaults are good for debugging!
- However, they do not always happen.

- Uninitialized read

```
int a, b;
a = b; // NO SEGFAULT
```

- Reading the heap

```
int* a = new int [5];
int b = a[7]; // USUALLY NO SEGFAULT
```

- Writing the heap

```
int* a = new int [5];
a[7] = 14;     // MAYBE NO SEGFAULT
```

- Using freed memory

```
int* a = new int [5];
delete [] a;
int b = a[3]; // USUALLY NO SEGFAULT
```

- Leaking memory

```
int* a = new int [5];
a = new int [10]; // NO SEGFAULT
a = nullptr;      // NO SEGFAULT
```

Does it Segfault?

- Segfaults are good for debugging!
- However, they do not always happen.

- Uninitialized read

```
int a, b;
a = b; // NO SEGFAULT
```

- Reading the heap

```
int* a = new int [5];
int b = a[7]; // USUALLY NO SEGFAULT
```

- Writing the heap

```
int* a = new int [5];
a[7] = 14;     // MAYBE NO SEGFAULT
```

- Using freed memory

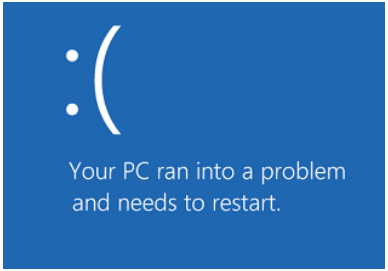
```
int* a = new int [5];
delete [] a;
int b = a[3]; // USUALLY NO SEGFAULT
```

- Leaking memory

```
int* a = new int [5];
a = new int [10]; // NO SEGFAULT
a = nullptr;      // NO SEGFAULT
```

- Freeing unallocated memory

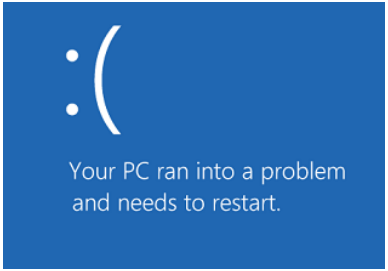
```
int* a;
free(a); // USUALLY NO SEGFAULT
```



Memory errors are:

- hard to find

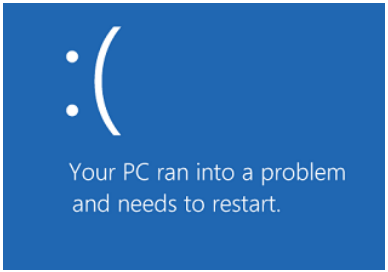
Good at Hiding



Memory errors are:

- hard to find
- often only appear occasionally

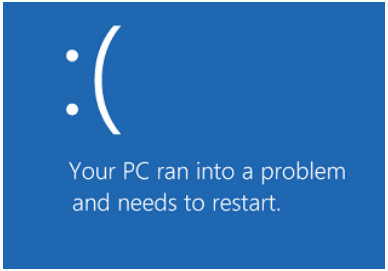
Good at Hiding



Memory errors are:

- hard to find
- often only appear occasionally
- cause bugs in different piece of code

Good at Hiding



Memory errors are:

- hard to find
- often only appear occasionally
- cause bugs in different piece of code
- happen easily and frequently

There are tools that can help you find them:

- Valgrind

There are tools that can help you find them:

- Valgrind
- address sanitizer (in GCC/Clang)

There are tools that can help you find them:

- Valgrind
- address sanitizer (in GCC/Clang)
- memory sanitizer (in GCC/Clang)

Finding Memory Errors

There are tools that can help you find them:

- Valgrind
- address sanitizer (in GCC/Clang)
- memory sanitizer (in GCC/Clang)
- Electric fence

There are tools that can help you find them:

- Valgrind
- address sanitizer (in GCC/Clang)
- memory sanitizer (in GCC/Clang)
- Electric fence
- dmalloc

Address Sanitizer

Address Sanitizer

Address Sanitizer (ASan)...

Address Sanitizer

Address Sanitizer (ASan)...

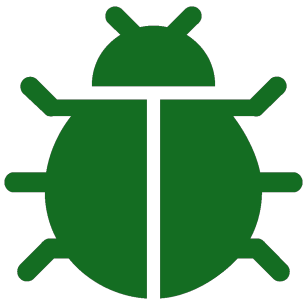
- is a compiler extension implemented in GCC and Clang
- is activated with compiler flag `-fsanitize=address`

Address Sanitizer

Address Sanitizer (ASan)...

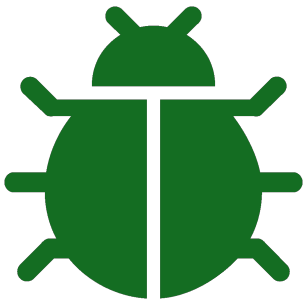
- is a compiler extension implemented in GCC and Clang
- is activated with compiler flag `-fsanitize=address`
- instruments code to detect memory errors at runtime
- should only be used as debugging tool

What Does it Find?



ASan detects

What Does it Find?



ASan detects

- Out-of-bounds accesses to heap, stack and globals

What Does it Find?

ASan detects

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return
- Use-after-scope

What Does it Find?

ASan detects

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return
- Use-after-scope
- Double-free, invalid free

What Does it Find?

ASan detects

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return
- Use-after-scope
- Double-free, invalid free
- Memory leaks (experimental)

Example

```
typedef struct {
    void (*print)(char*);
} operation;

int main(int argc, char* argv[]) {
    operation* io = (operation*)malloc(sizeof(operation));
    io->print = puts;
    io->print("Hallo ");
    free(io);

    if(argc > 1) {
        char* buffer = (char*)malloc(8);
        strncpy(buffer, argv[1], 7);
        io->print(buffer);
        free(buffer);
    }
    return 0;
}
```

```
Hallo
=================================================================
==27728==ERROR: AddressSanitizer: heap-use-after-free on address 0x60200000eff0 at pc 0x00000040094d bp 0x7fffffffdcd0 sp 0x7fffffffdcc0
READ of size 8 at 0x60200000eff0 thread T0
   #0 0x40094c in main /home/mschwarz/Teaching/IIS/asan.c:24
   #1 0x7ffff6ac182f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2082f)
   #2 0x4007f8 in _start (/home/mschwarz/Teaching/IIS/a.out+0x4007f8)

0x60200000eff0 is located 0 bytes inside of 8-byte region [0x60200000eff0,0x60200000eff8)
freed by thread T0 here:
   #0 0x7ffff6f022ca in __interceptor_free (/usr/lib/x86_64-linux-gnu/libasan.so.2+0x982ca)
   #1 0x4008f7 in main /home/mschwarz/Teaching/IIS/asan.c:19
   #2 0x7ffff6ac182f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2082f)

previously allocated by thread T0 here:
   #0 0x7ffff6f02602 in malloc (/usr/lib/x86_64-linux-gnu/libasan.so.2+0x98602)
   #1 0x4008e2 in main /home/mschwarz/Teaching/IIS/asan.c:16
   #2 0x7ffff6ac182f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2082f)

SUMMARY: AddressSanitizer: heap-use-after-free /home/mschwarz/Teaching/IIS/asan.c:24 main
Shadow bytes around the buggy address:
  0x0c047fff9da0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff9db0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff9dc0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff9dd0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff9de0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x0c047fff9df0: fa fa fa fa fa fa fa fa 00 fa fa fa[fd]fa
  0x0c047fff9e00: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff9e10: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff9e20: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff9e30: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff9e40: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
 Addressable:           00
 Partially addressable: 01 02 03 04 05 06 07
 Heap left redzone:       fa
 Heap right redzone:      fb
 Freed heap region:       fd
 Stack left redzone:      f1
 Stack mid redzone:       f2
 Stack right redzone:     f3
 Stack partial redzone:   f4
 Stack after return:      f5
 Stack use after scope:   f8
 Global redzone:          f9
 Global init order:       f6
 Poisoned by user:        f7
 Container overflow:      fc
 Array cookie:            ac
 Intra object redzone:    bb
 ASan internal:           fe
==27728==ABORTING
```

How Does it Work?

Address sanitizer introduces shadow memory:

How Does it Work?

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word

How Does it Work?

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:

How Does it Work?

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
  - `00` → all 8 bytes are addressable
  - `01-07` → first 1-7 bytes are addressable
  - `80-ff` → none of the bytes are addressable

How Does it Work?

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
  - `00` → all 8 bytes are addressable
  - `01-07` → first 1-7 bytes are addressable
  - `80-ff` → none of the bytes are addressable
- Check the shadow memory on each access

How Does it Work?

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
  - `00` → all 8 bytes are addressable
  - `01-07` → first 1-7 bytes are addressable
  - `80-ff` → none of the bytes are addressable
- Check the shadow memory on each access
- Allocate and update shadow memory as needed

How Does it Work?

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
  - `00` → all 8 bytes are addressable
  - `01-07` → first 1-7 bytes are addressable
  - `80-ff` → none of the bytes are addressable
- Check the shadow memory on each access
- Allocate and update shadow memory as needed

- Here is an example

```
#include <stdlib.h>
int main(int argc, char* argv[])
{
  // argc is 1
  int* a = malloc(22);
  a[0*argc] = 5;
  a[4*argc] = 42;
  a[3*argc] = 15;
  a[5*argc] = 18;
  free(a);
  return 0;
}
// Shadow memory:
// fa fa fa fa fa fa fa fa
// fa fa fa fa fa fa fa fa
// fa fa fa fa fa fa fa fa
```

How Does it Work?

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
  - `00` → all 8 bytes are addressable
  - `01-07` → first 1-7 bytes are addressable
  - `80-ff` → none of the bytes are addressable
- Check the shadow memory on each access
- Allocate and update shadow memory as needed

- Here is an example

```
#include <stdlib.h>
int main(int argc, char* argv[])
{
  // argc is 1
  int* a = malloc(22);// <--
  a[0*argc] = 5;
  a[4*argc] = 42;
  a[3*argc] = 15;
  a[5*argc] = 18;
  free(a);
  return 0;
}
// Shadow memory:
// fa fa fa fa fa fa fa fa
// fa fa fa fa[00 00 06]fa
// fa fa fa fa fa fa fa fa
```

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
  - `00` → all 8 bytes are addressable
  - `01-07` → first 1-7 bytes are addressable
  - `80-ff` → none of the bytes are addressable
- Check the shadow memory on each access
- Allocate and update shadow memory as needed

- Here is an example

```
#include <stdlib.h>
int main(int argc, char* argv[])
{
  // argc is 1
  int* a = malloc(22);
  a[0*argc] = 5;  // <--
  a[4*argc] = 42;
  a[3*argc] = 15;
  a[5*argc] = 18;
  free(a);
  return 0;
}
// Shadow memory:
// fa fa fa fa fa fa fa fa
// fa fa fa fa[00]00 06 fa
// fa fa fa fa fa fa fa fa
```

How Does it Work?

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
  - `00` → all 8 bytes are addressable
  - `01-07` → first 1-7 bytes are addressable
  - `80-ff` → none of the bytes are addressable
- Check the shadow memory on each access
- Allocate and update shadow memory as needed

- Here is an example

```
#include <stdlib.h>
int main(int argc, char* argv[])
{
  // argc is 1
  int* a = malloc(22);
  a[0*argc] = 5;
  a[4*argc] = 42; // <--
  a[3*argc] = 15;
  a[5*argc] = 18;
  free(a);
  return 0;
}
// Shadow memory:
// fa fa fa fa fa fa fa fa
// fa fa fa fa 00 00[06]fa
// fa fa fa fa fa fa fa fa
```

How Does it Work?

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
  - `00` → all 8 bytes are addressable
  - `01-07` → first 1-7 bytes are addressable
  - `80-ff` → none of the bytes are addressable
- Check the shadow memory on each access
- Allocate and update shadow memory as needed

- Here is an example

```
#include <stdlib.h>
int main(int argc, char* argv[])
{
  // argc is 1
  int* a = malloc(22);
  a[0*argc] = 5;
  a[4*argc] = 42;
  a[3*argc] = 15; // <--
  a[5*argc] = 18;
  free(a);
  return 0;
}
// Shadow memory:
// fa fa fa fa fa fa fa fa
// fa fa fa fa 00[00]06 fa
// fa fa fa fa fa fa fa fa
```

How Does it Work?

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
  - `00` → all 8 bytes are addressable
  - `01-07` → first 1-7 bytes are addressable
  - `80-ff` → none of the bytes are addressable
- Check the shadow memory on each access
- Allocate and update shadow memory as needed

- Here is an example

```
#include <stdlib.h>
int main(int argc, char* argv[])
{
  // argc is 1
  int* a = malloc(22);
  a[0*argc] = 5;
  a[4*argc] = 42;
  a[3*argc] = 15;
  a[5*argc] = 18; // <-- ERROR
  free(a);
  return 0;
}
// Shadow memory:
// fa fa fa fa fa fa fa fa
// fa fa fa fa 00 00[06]fa
// fa fa fa fa fa fa fa fa
```

How Does it Work?

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
  - `00` → all 8 bytes are addressable
  - `01-07` → first 1-7 bytes are addressable
  - `80-ff` → none of the bytes are addressable
- Check the shadow memory on each access
- Allocate and update shadow memory as needed

- Here is an example

```
#include <stdlib.h>
int main(int argc, char* argv[])
{
  // argc is 1
  int* a = malloc(22);
  a[0*argc] = 5;
  a[4*argc] = 42;
  a[3*argc] = 15;
  a[5*argc] = 18;
  free(a);         // <--
  return 0;
}
// Shadow memory:
// fa fa fa fa fa fa fa fa
// fa fa fa fa[fd fd fd]fa
// fa fa fa fa fa fa fa fa
```

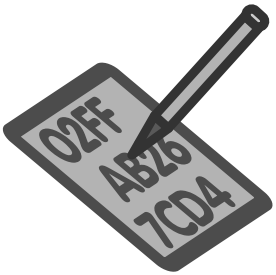
Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
  - `00` → all 8 bytes are addressable
  - `01-07` → first 1-7 bytes are addressable
  - `80-ff` → none of the bytes are addressable
- Check the shadow memory on each access
- Allocate and update shadow memory as needed

- Here is an example

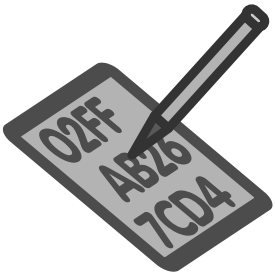
```
#include <stdlib.h>
int main(int argc, char* argv[])
{
  // argc is 1
  int* a = malloc(22);
  a[0*argc] = 5;
  a[4*argc] = 42;
  a[3*argc] = 15;
  a[5*argc] = 18;
  free(a);
  return 0;        // <--
}
// Shadow memory:
// fa fa fa fa fa fa fa fa
// fa fa fa fa fd fd fd fa
// fa fa fa fa fa fa fa fa
```

- `malloc` and `free` are replaced



- `malloc` and `free` are replaced
- Memory around `malloc`'d segments is poisened

```
// ff ff fa fa fa 00 00 00 04 fa fa fa ff ff
//      [red zone] allocated [red zone]
```



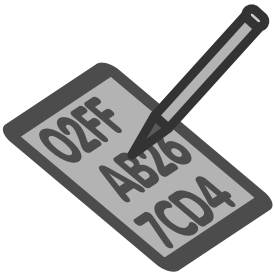
- `malloc` and `free` are replaced
- Memory around `malloc`'d segments is poisened

```
// ff ff fa fa fa 00 00 00 04 fa fa fa ff ff
//      [red zone] allocated [red zone]
```

- `free`'d memory is poisened and moved to quarantine

```
// ff ff fa fa fa fd fd fd fd fa fa fa ff ff
//       red zone[ quarantine]red zone
```

Instrumentation Overview



- `malloc` and `free` are replaced
- Memory around `malloc`'d segments is poisened

```
// ff ff fa fa fa 00 00 00 04 fa fa fa ff ff
//      [red zone] allocated [red zone]
```

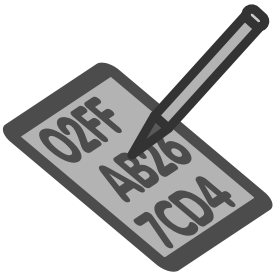
- `free`'d memory is poisened and moved to quarantine

```
// ff ff fa fa fa fd fd fd fd fa fa fa ff ff
//       red zone[ quarantine]red zone
```

- Every memory access is first checked if poisoned

```
if (IsPoisoned(address)) {
  ReportError(address, kAccessSize, kIsWrite);
}
*address = ...;  // or: ... = *address;
```

Instrumentation Overview



- `malloc` and `free` are replaced
- Memory around `malloc`'d segments is poisened

```
// ff ff fa fa fa 00 00 00 04 fa fa fa ff ff
//      [red zone] allocated [red zone]
```

- `free`'d memory is poisened and moved to quarantine

```
// ff ff fa fa fa fd fd fd fd fa fa fa ff ff
//       red zone[ quarantine]red zone
```

- Every memory access is first checked if poisoned

```
if (IsPoisoned(address)) {
  ReportError(address, kAccessSize, kIsWrite);
}
*address = ...;  // or: ... = *address;
```

- red zones are also added to stack allocation and global arrays

Limitations

- Slowdown of approximately factor 2
- Increased memory usage of factor 2 to 5, depending on allocations
- Cannot prevent all arbitrary memory corruptions
- Does not protect against integer overflows
- Adjacent buffers in structs/classes are not protected

Leak Sanitizer ...

- part of ASan, but also available as standalone

Leak Sanitizer

Leak Sanitizer ...

- part of ASan, but also available as standalone
- implemented in GCC and Clang
- is activated with compiler flag `-fsanitize=leak`

Leak Sanitizer ...

- part of ASan, but also available as standalone
- implemented in GCC and Clang
- is activated with compiler flag `-fsanitize=leak`
- detects memory leaks at runtime

Leak Sanitizer

Leak Sanitizer ...

- part of ASan, but also available as standalone
- implemented in GCC and Clang
- is activated with compiler flag `-fsanitize=leak`
- detects memory leaks at runtime
- should only be used as debugging tool

```
char* buffer;
void print(const char* tag,const char* info)
{
  buffer = malloc(strlen(tag) + strlen(info) + 4);
  strcpy(buffer,tag);
  strcat(buffer,": ");
  strcat(buffer,info);
  strcat(buffer,"\n");
  puts(buffer);
  free(buffer);
}
int main(int argc, char* argv[]) {
    buffer = malloc(32);
    fgets(buffer,32,stdin);
    const char* tag = strdup("user input");
    print(tag,buffer);
    return 0;
}
```

Leak Sanitizer

```
hello world
user input: hello world



=================================================================
==20366==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 32 byte(s) in 1 object(s) allocated from:
    #0 0x7f3a3bfb8fee in __interceptor_malloc (/usr/lib/x86_64-linux-gnu/liblsan.
    #1 0x559f9420fa80 in main /home/dgruss/buffer.c:17
    #2 0x7f3a3bc053f0 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x203

Direct leak of 11 byte(s) in 1 object(s) allocated from:
    #0 0x7f3a3bfb8fee in __interceptor_malloc (/usr/lib/x86_64-linux-gnu/liblsan.
    #1 0x7f3a3bc72799 in __strdup (/lib/x86_64-linux-gnu/libc.so.6+0x8d799)

SUMMARY: LeakSanitizer: 43 byte(s) leaked in 2 allocation(s).
```

Memory Sanitizer

Memory Sanitizer

Memory sanitizer (MSan) ...

- is not part of ASan, but similar interface

Memory Sanitizer

Memory sanitizer (MSan) ...

- is not part of ASan, but similar interface
- detects use of uninitialized values

Memory Sanitizer

Memory sanitizer (MSan) ...

- is not part of ASan, but similar interface
- detects use of uninitialized values
- implemented in GCC and Clang
- is activated with compiler flag `-fsanitize=memory`

Memory Sanitizer

Memory sanitizer (MSan) ...

- is not part of ASan, but similar interface
- detects use of uninitialized values
- implemented in GCC and Clang
- is activated with compiler flag `-fsanitize=memory`
- aims at replacing valgrind memcheck tool

Memory Sanitizer

Memory sanitizer (MSan) ...

- is not part of ASan, but similar interface
- detects use of uninitialized values
- implemented in GCC and Clang
- is activated with compiler flag `-fsanitize=memory`
- aims at replacing valgrind memcheck tool

Memory Sanitizer Details

- Implements a bit precise shadow memory tracking initialization
- Each operation on normal memory has similar operation:
  - $A = B \ \& \ C \quad \rightarrow \quad A' = (B' \& C') \mid (B \& C') \mid (B' \& C)$
  - $A = B \mid C \quad \rightarrow \quad A' = (B' \& C') \mid (\sim B \& C') \mid (B' \& \sim C)$
  - ...
- Copying and use of undefined memory are generally allowed
- Reports a bug whenever undefined value influences control flow
- Typical slowdown: $3\times$

Memory Sanitizer Details

- Implements a bit precise shadow memory tracking initialization
- Each operation on normal memory has similar operation:
  - $A = B \ \& \ C \ \rightarrow \ A' = (B' \& C') \mid (B \& C') \mid (B' \& C)$
  - $A = B \mid C \ \rightarrow \ A' = (B' \& C') \mid (\sim B \& C') \mid (B' \& \sim C)$
  - ...
- Copying and use of undefined memory are generally allowed
- Reports a bug whenever undefined value influences control flow
- Typical slowdown: $3\times$
- Memory impact: $2 - 3\times$

Memory Sanitizer Details

- Implements a bit precise shadow memory tracking initialization
- Each operation on normal memory has similar operation:
  - $A = B \ \& \ C \ \rightarrow \ A' = (B' \& C') \mid (B \& C') \mid (B' \& C)$
  - $A = B \mid C \ \rightarrow \ A' = (B' \& C') \mid (\sim B \& C') \mid (B' \& \sim C)$
  - ...
- Copying and use of undefined memory are generally allowed
- Reports a bug whenever undefined value influences control flow
- Typical slowdown: $3\times$
- Memory impact: $2 - 3\times$
- reserves a 64 TB region virtual address space for shadows

Memory Sanitizer - Example

```
int main(int argc, char** argv) {
  int* a = new int[10];
  a[5] = 0;
  volatile int b = a[argc];
  if (b)
    printf("xx\n");
  return 0;
}
```

Memory Sanitizer - Example 2

```
==25491==WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x563fa6533efe  (/home/dgruss/a.out+0x93efe)
    #1 0x7fd1cbbd83f0  (/lib/x86_64-linux-gnu/libc.so.6+0x203f0)
    #2 0x563fa64baff9  (/home/dgruss/a.out+0x1aff9)

  Uninitialized value was stored to memory at
    #0 0x563fa6533e73  (/home/dgruss/a.out+0x93e73)
    #1 0x7fd1cbbd83f0  (/lib/x86_64-linux-gnu/libc.so.6+0x203f0)

  Uninitialized value was created by a heap allocation
    #0 0x563fa65312a0  (/home/dgruss/a.out+0x912a0)
    #1 0x563fa6533c58  (/home/dgruss/a.out+0x93c58)
    #2 0x7fd1cbbd83f0  (/lib/x86_64-linux-gnu/libc.so.6+0x203f0)

SUMMARY: MemorySanitizer: use-of-uninitialized-value (/home/dgruss/a.ou
Exiting
```

Questions?


99 little bugs in the code.
99 little bugs in the code.
Take one down, patch it around.
127 little bugs in the code...