D as Better C Compiler

by Walter Bright
dlang.org

C

- Brilliantly conceived language
- Major force for 40 years
- Engine for major critical software
- Well known and understood
- “Man behind the curtain”

All Is Not Roses

- 40 years of advancement in programming languages
- No memory safety

Memory Safety

“being protected from various software bugs and security vulnerabilities when dealing with memory access, such as buffer overflows and dangling pointers.”

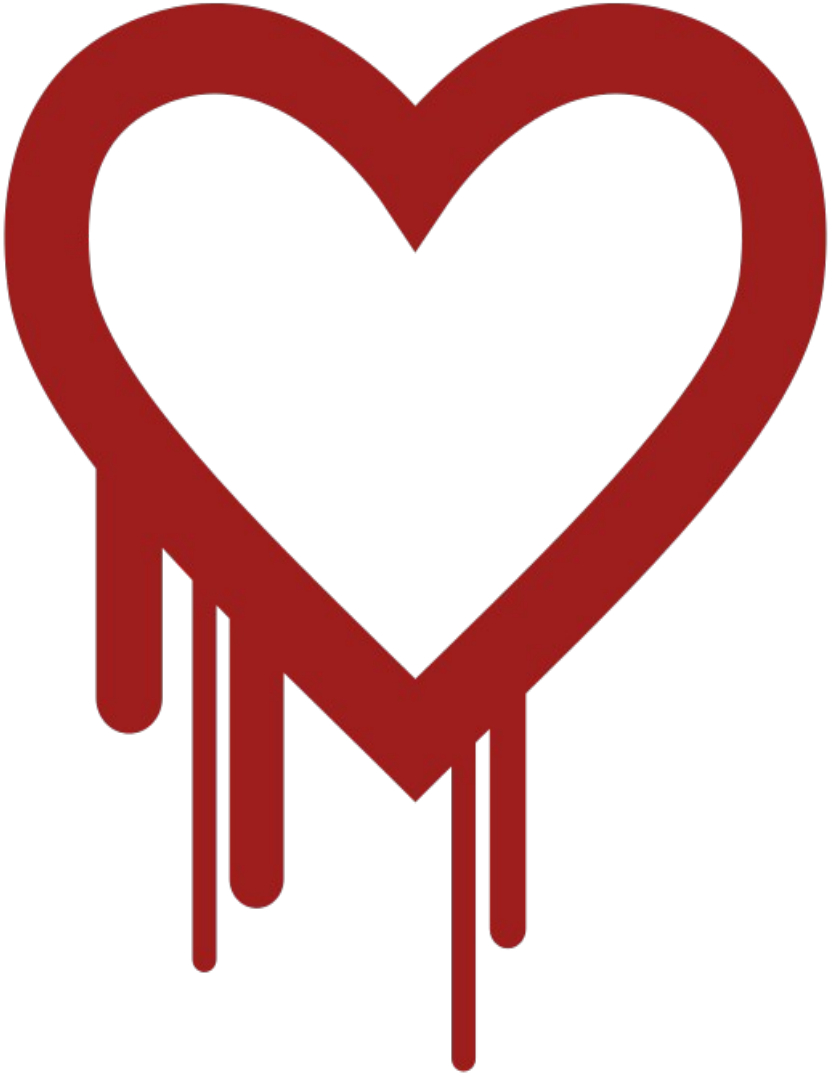
– wikipedia

Costs of Memory Unsafety

- millions of dollars
- endless hours of effort to find and correct
- constant threat
- embarrassment

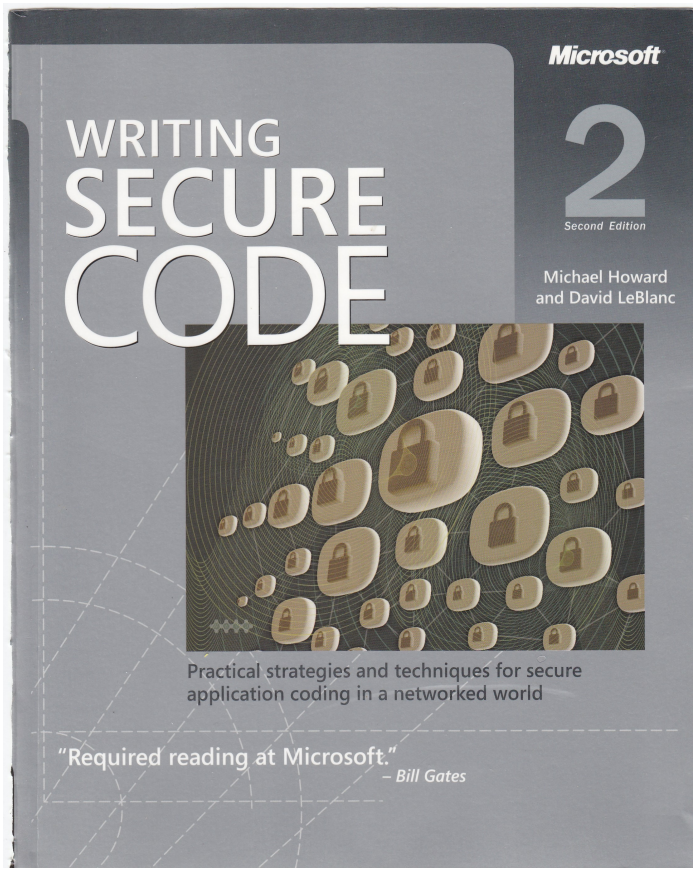
Buffer Overflow

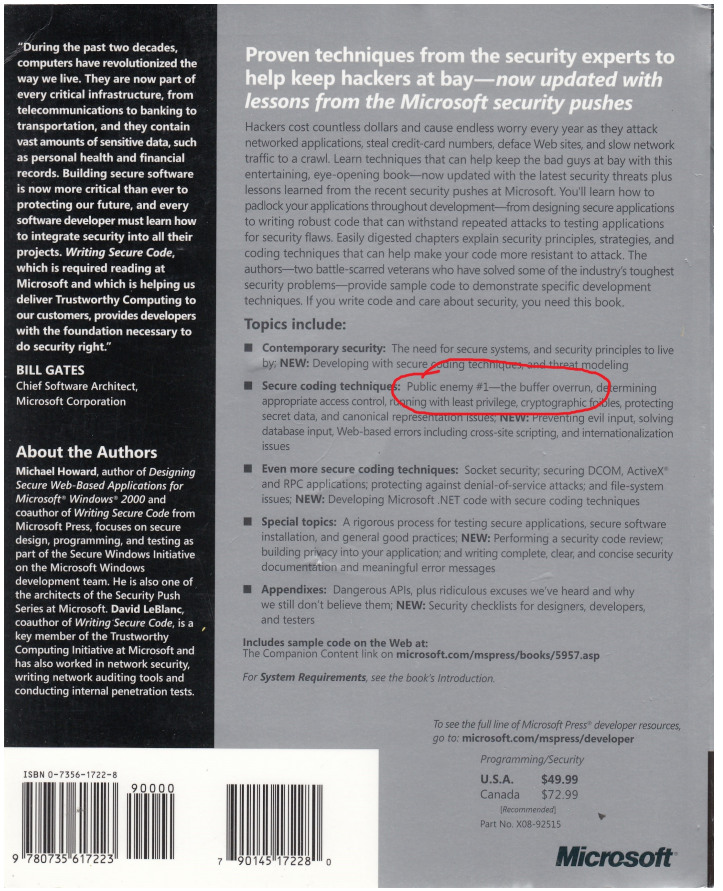
Heartbleed



https://en.wikipedia.org/wiki/Heartbleed

Buffer Overflow





Public Enemy #1

Microsoft's Writing Secure Code 2

Buffer Overflow

Morris Worm

First internet worm relied on buffer overflow C bug

Buffer Overflow

#3 on list of top 25 security vulnerabilities

http://cwe.mitre.org/top25/index.html#CWE-120

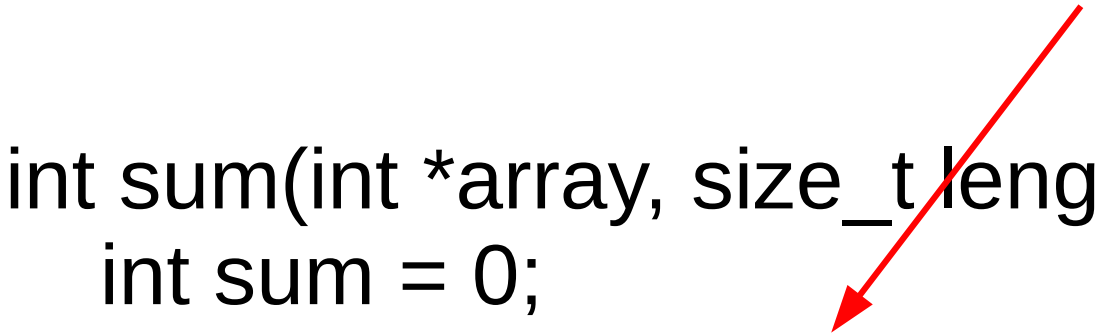
Not a Buffer Overflow

Good News!

It's not your fault

- Has persisted for 40 years
- Not a tractable problem
- Not fixable by fixing the programmer

It's a Tooling Problem



```
int sum(int *array, size_t length) {
    int sum = 0;
    for (size_t i = 0; i <= length; ++i)
        sum += array[i];
    return sum;
}
```

– Wizard of Oz

D Arrays are Phat Pointers

```
struct Array {
    size_t length;
    int* ptr;
}
```

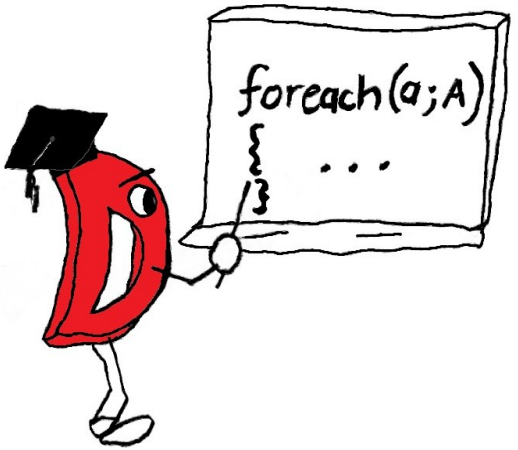
https://dlang.org/spec/arrays.html#dynamic-arrays

First D Version

```
int sum(int[ ] array)
{
    int sum = 0;
    for (size_t i = 0; i < array.length; ++i)
        sum += array[i];
    return sum;
}
```

Using foreach

```
int sum(int[ ] array)
{
    int sum = 0;
    foreach (i; 0 .. array.length)
        sum += array[i];
    return sum;
}
```



More Advanced foreach

```
int sum(int[ ] array)
{
    int sum = 0;
    foreach (value; array)
        sum += value;
    return sum;
}
```

Same problem with 0 terminated strings

D strings are just arrays of characters

And Much More Safety

- no uninitialized pointers
- no pointers to expired stack frames
- no aliasing pointers with other types
- no implicit narrowing conversions
- pointer lifetime tracking
- etc...

Calling C Code from D

```
extern (C) void* malloc(size_t);

void allocArray(T)(size_t numElements)
{
    auto p = malloc(T.sizeof * numElements);
    assert(p);
    return p[0 .. numElements];
}
```

Problem

D code needs to link with the D runtime library. But an existing C project does not link with the D runtime library. Linking it in produces issues with the size of the D runtime library, and how/when it is initialized compared to that of the existing C project.

Solution - BetterC

Create a subset of D that does not require the D runtime library.

Features Altered

- assert() failures now go to C standard library
- RAII no longer unwinds exceptions
- No dynamic type info
- No exception handling
- No automatic memory management

What's this going to cost?

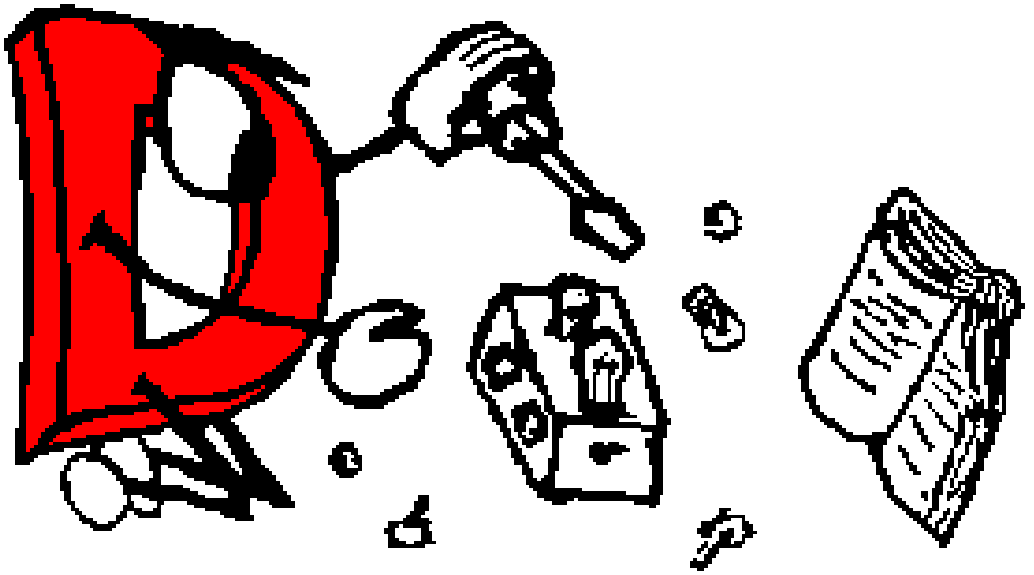
C

```
#include <stdio.h>

int main(size_t argc, char** argv)
{
    for (size_t i = 0; i < argc; ++i)
        printf("arg[%zd] = %s\n", i, argv[i]);
    return 0;
}
```

D as Better C

```
import core.stdc.stdio;

extern (C)
int main(size_t argc, char** argv)
{
    foreach (i, s; argv[0 .. argc])
        printf("arg[%zd] = %s\n", i, s);
    return 0;
}
```

Some Assembly Required



```
    sub ESP,01Ch
    mov 014h[ESP],ESI
    mov ESI,020h[ESP]
    mov 018h[ESP],EDI
    mov EDI,024h[ESP]
    mov 010h[ESP],EBX
    xor EBX,EBX
    test ESI,ESI
    je  L1
L2: mov ECX,[EBX*4][EDI]
    mov 8[ESP],ECX
    mov 4[ESP],EBX
    mov [ESP],offset _DATA
    call _printf
    inc EBX
    cmp EBX,ESI
    jb  L2
L1: mov EBX,010h[ESP]
    mov ESI,014h[ESP]
    mov EDI,018h[ESP]
    add ESP,01Ch
    xor EAX,EAX
    ret
```

```
    sub ESP,01Ch
    mov 014h[ESP],ESI
    mov ESI,020h[ESP]
    mov 018h[ESP],EDI
    mov EDI,024h[ESP]
    mov 010h[ESP],EBX
    xor EBX,EBX
    test ESI,ESI
    je  L1
L2: mov ECX,[EBX*4][EDI]
    mov 8[ESP],ECX
    mov 4[ESP],EBX
    mov [ESP],offset CONST
    call _printf
    inc EBX
    cmp EBX,ESI
    jb  L2
L1: mov EBX,010h[ESP]
    mov ESI,014h[ESP]
    mov EDI,018h[ESP]
    add ESP,01Ch
    xor EAX,EAX
    ret
```

No Compromise

The same code is generated!

How to compile and link in a BetterC function into a C project...

The Original C Code

```
#include <stdio.h>
int sum(int *array, size_t length) {
  int sum = 0;
  for (size_t i = 0; i < length; ++i)
        sum += array[i];
  return sum;
}
int main(size_t argc, char** argv) {
    int array[3] = { 1, 37, 28 };
    int s = sum(array,sizeof(array)/sizeof(array[0]));
    printf("sum = %d\n", s);
    return 0;
}
```

main.c

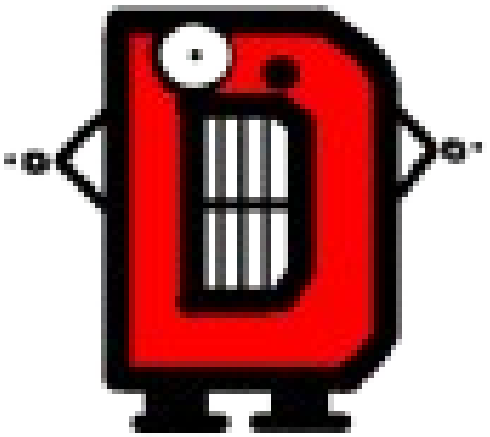
```
#include <stdio.h>

int sum(int *array, size_t length);

int main(size_t argc, char** argv) {
    int array[3] = { 1, 37, 28 };
    int s = sum(array,sizeof(array)/sizeof(array[0]));
    printf("sum = %d\n", s);
    return 0;
}
```

sum.d

```
extern ( C )
int sum(int *array, size_t length) {
    return sum(array[0 .. length]);
}

@safe int sum(int[ ] array) {
    int sum = 0;
    foreach (value; array)
        sum += value;
    return sum;
}
```

Compiling

dmd -c sum.d
dmc main.c sum.obj



Converting a C file to D

- Remove preprocessor metaprogramming
  - (should get rid of it anyway)
- Copy the code to a .d file
- Translate one function at a time
  - run the test suite after each

Don't Refactor/Improve/Fix

- Wait until it is all translated
  - and passes all tests!
  - resist overwhelming temptation
    - or you'll be sorry!
- Just rote translate it

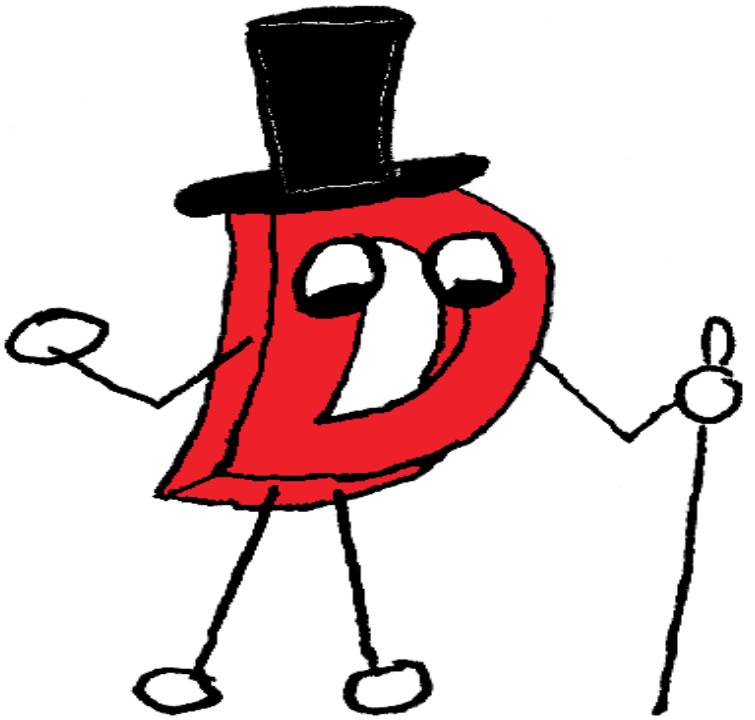
Proof That This Works

The Digital Mars C/C++ Compiler Front End

https://github.com/DigitalMars/Compiler/tree/master/dm/src/dmd

Now That It's Working in D

- What can be done to benefit?
  - (Besides using proper arrays!)



```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

bool see(void* buf, int maybe);

bool isPossible(int maybe) {
  void *buf = malloc(100);
  if (see(buf, maybe)) {
    free(buf);
    return true;
  }
  free(buf);
  return false;
}
```

```
import core.stdc.stdlib;  // modules!

bool see(void* buf, int maybe);

bool isPossible(int maybe) {
  void *buf = malloc(100);
  scope(exit) free(buf);  // scope guard
  if (see(buf, maybe))
    return true;
  return false;
}
```

RAII

```
import core.stdc.stdlib;

bool see(void* buf, int maybe);

struct S { void* buf; ~this() { free(buf); } }

bool isPossible(int maybe) {
  S s;
  s.buf = malloc(100);
  if (see(s.buf, maybe))
    return true;
  return false;
}
```

```
int compute(int x, int y) {
    if (x == y)
        return -1; // error
    if (x == 5)
        return 2;
    if (x == y + 3)
        return 7;
    return -1; // error
}
```

```
int compute(int x, int y) {
    const bool log = true;
    if (x == y)
    { if (log) printf("fail\n"); return -1; }
    if (x == 5)
    { if (log) printf("success 2\n"); return 2; }
    if (x == y + 3)
    { if (log) printf("success 7\n"); return 7; }
    if (log) printf("fail\n");
    return -1;
}
```

Nested Functions

```
int compute(int x, int y) {
    enum log = true;
    int fail() {
        if (log) printf("fail\n"); return -1;
    }
    int success(int i) {
        if (log) printf("success %d\n", i); return i;
    }

    if (x == y)
        return fail();
    if (x == 5)
        return success(2);
    if (x == y + 3)
        return success(7);
    return fail();
}
```

Some Things D Can't Do

```
#define BEGIN {
#define END     }

int sum(int *array, size_t length)
  BEGIN
    int sum = 0;
    size_t i;
    for (i = 0; i < length; ++i)
      BEGIN
        sum += array[i];
      END
    return sum;
  END
```

Get Started Today!

- Incrementally use D functions in large C project
- Stop buffer overflows and other safety bugs
- Use D to improve code

dlang.org