

Security bug of the week?

SECTIONS ▾ **FORUM** | **SUBSCRIBE** |   | SIGN IN

 AN UPDATE UH-OH

Owners mourn spoiled food after firmware update bricks Samsung smart fridges

“So absurd and embarrassing ...”

SCHARON HARDING – 23 SEPT 2026 19:39 | 192



→ A marketing image for Samsung's Bespoke AI refrigerators.
Credit: Samsung

But is this really a *security* problem? There is no attacker here...

Security bug of the week - CVE pending

Pre-auth format string bug in tac_plus implementation of TACACS+ can lead to **RCE** and **DoS**

Bug

```
/*
 * Send an authentication reply packet indicating an error has occurred.
 * msg is a null terminated character string
 */
void
send_authen_error(char *msg)
{
    char buf[NI_MAXHOST + 256];

    if (snprintf(buf, sizeof(buf), "%s %s: %s", session.peer, session.port,
                msg) == -1)
        strcpy(buf, "");
    report(LOG_ERR, buf);
    send_authen_reply(TAC_PLUS_AUTHEN_STATUS_ERROR, buf, strlen(buf), NULL, 0,
                    0);
}
```

safe snprintf using a format string

unsafe if buf contains format characters

Fix

```
packet.c:238  report(LOG_ERR, buf);    ->  report(LOG_ERR, "%s", buf);
packet.c:61   report(LOG_ERR, msg);   ->  report(LOG_ERR, "%s", msg);
```

<https://www.elttam.com/blog/att-cking-tacacs-to-pwn-your-network-via-a-pre-auth-rce>

What the hell is TACACS+?



<https://www.youtube.com/watch?v=S5oPIP4TRfY>

Affected systems - FOSS

Distro / Source	Package	Fork	Version	Status
ALT Linux p10 / p11	tac_plus	Shrubbery	4.0.4.28-alt2	Vulnerable - await update
ALT Linux sisyphus	tac_plus-ng	Huber	2026.06.17.9-alt1	Not affected
Arch (AUR)	tacacs-plus	Shrubbery	4.0.4.28-1	Vulnerable - orphaned
Debian	tacacs+	Shrubbery	4.0.4.27a-3	Vulnerable (EOL only)
deepin 20	tacacs+	Unconfirmed	?	Unconfirmed - verify
EPEL 8 / 9	tacacs	Meta	F4.0.4.28.7fb~20231005g4fdf178	Vulnerable - no upstream fix
Fedora 35-42	tacacs	Meta	F4.0.4.28.7fb (g25fd8f0)	Vulnerable - no upstream fix
FreeBSD 14 / 15	net/tac_plus4	Shrubbery	F4.0.4.28_2	Vulnerable - await F4.0.4.32 port
Mageia 10 / cauldron	tac_plus	Shrubbery	4.0.4.28-2	Vulnerable - await update
NetBSD pkgsrc	net/tacacs	Shrubbery	4.0.4.28nb9	Vulnerable - await update
nixpkgs	tac_plus	Unconfirmed	4.0.4.31	Vulnerable - await update
OpenBSD Ports	tac_plus	Shrubbery	4.0.4.28 rev 4	Vulnerable - await update
OpenPKG	tac_plus	Unconfirmed	?	Unconfirmed - verify
openSUSE (Tumbleweed)	tac_plus	Shrubbery	4.0.4.28	Vulnerable - await update
OpenWrt 23.05 / 24.10 / 25.12	tac_plus (+ - pam)	Meta	4.0.4.28-r2/r3/r4 (snapshot 2019.02.11)	Vulnerable - no upstream fix
PureOS amber	tacacs+	Unconfirmed	?	Unconfirmed - verify
Rosa 2021.1 / 13	tac_plus	Unconfirmed	?	Unconfirmed - verify
Ubuntu	tacacs+	Shrubbery	4.0.4.27a-3	Vulnerable (EOL/ESM only)

<https://ifin.network/t/cve-pending-pre-auth-format-string-bug-in-tac-plus-shrubbery-networks/859>

Affected systems - commercial

Vendor	Product	Fork	Version	Status
Cisco	ISE (Device Admin license)	?	All	Not affected per Cisco PSIRT
Cisco	Secure ACS (EOL)	?	All	Not affected per Cisco PSIRT
DialWays	DialWays AAA	?	?	?
Fortinet	FortiAuthenticator	?	?	?
Fortinet	FortiGate / FortiOS	n/a	n/a	client-only?
HPE Aruba	ClearPass Policy Manager	?	?	?
Juniper	Junos	n/a	n/a	client-only?
Meta	Tacquito	none	?	Not affected?
NTTAcPlus	NTTAcPlus	?	?	?

<https://ifin.network/t/cve-pending-pre-auth-format-string-bug-in-tac-plus-shrubbery-networks/859>

Threat Intel

Validation / vulnerability detection

- draft YARA (validated against patched and unpatched packages, but not commercial):
[Yara for detecting tac_plus versions vulnerable to 2026-09 vulnerability · GitHub](#) ²
- scan for TCP port 49. If you can nmap, scan with NSE (benign, beyond stock Nmap, detects valid TACACS service and shared-secret status)
[Nmap NSE to detect valid TACACS and its shared-secret status \(benign fingerprint\) · GitHub](#)
- grep for strings starting with `tac` in package-listing output, including `tacplus` and `tac_plus`
- search filesystem for `tac_plus.conf`
- binwalk binaries for `Shrubbery`

<https://ifin.network/t/cve-pending-pre-auth-format-string-bug-in-tac-plus-shrubbery-networks/859>

Memory corruption

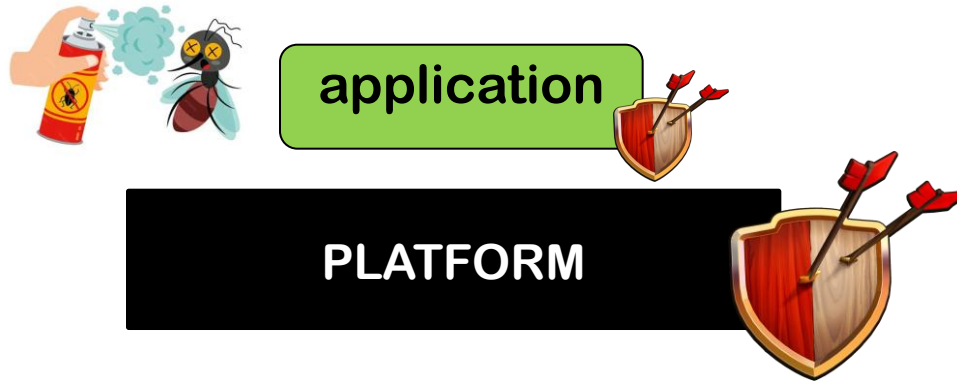
Application-level Countermeasures

Erik Poll

Digital Security

Radboud University Nijmegen

Application-level countermeasures



Last week: what can the compiler do against memory corruption problems?

*This week: **What can the programmer do?***

‘Generic’ application-level countermeasures



- **Dynamic Analysis** aka **DAST** (Dynamic Application Security Testing)
One very useful test technique: **fuzzing**



- **Static analysis** aka **SAST** (Static Application Security Testing)
By compiler **gcc -Wall**, **clang -Wall**, **msvs /analyze**
or dedicated SAST tool **ccpcheck**, **PRefast**, **Converity**, **SonarCube**, **CodeQL**, ...



- Using **safer coding practices**:
 - safer libraries
 - safer programming idioms
 - safer coding style
 - ...

These can be enforced with SAST

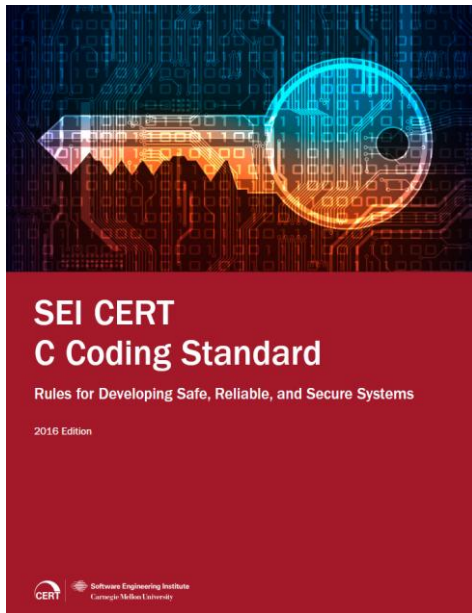


- **Switching to safer programming language**
such as **Rust**, **Swift** or **Go**

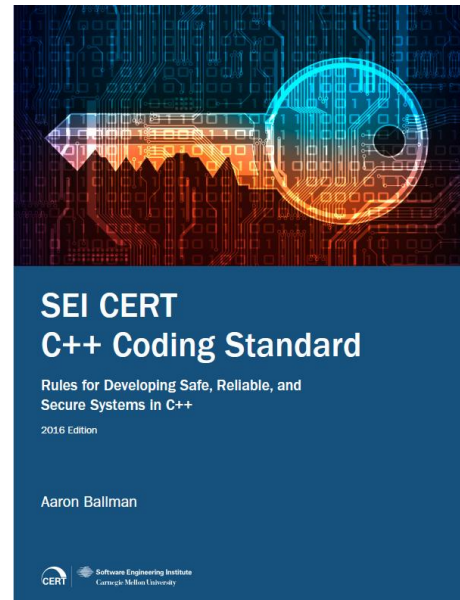
Or: switching from C to Objective C or C++



SEI CERT secure coding standards for C & C++



528 pages



429 pages

10 years old, so these won't contain latest insights

Free to download at <http://www.securecoding.cert.org>

Example best practices (from OpenSSF)

Run compiler with following flags

```
-O2 -Wall -Wformat -Wformat=2 -Wconversion -Wimplicit-fallthrough \  
-Werror=format-security -U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=3 \  
-D_GLIBCXX_ASSERTIONS -D_LIBCPP_HARDENING_MODE=_LIBCPP_HARDENING_MODE_FAST \  
-fstrict-flex-arrays=3 -fstack-clash-protection -fstack-protector-strong \  
-Wl,-z,nodlopen -Wl,-z,noexecstack -Wl,-z,relro -Wl,-z,now \  
-Wl,--as-needed -Wl,--no-copy-dt-needed-entries  
...
```

See eg <https://best.openssf.org/Compiler-Hardening-Guides/Compiler-Options-Hardening-Guide-for-C-and-C++.html>

Note: compilers can improve security in two ways:

1. **by more static checks**
2. **by inserting dynamic defensive checks in binary**
eg stack canaries, memory tagging, ENDBR instructions, ...

Application-specific countermeasures

Countermeasures tailored to a specific (part of a) application, as part of the higher level design

- Identify high-risk components and apply the generic countermeasures there – or there first, or more thoroughly
 - One common high-risk component: **PARSERS**
Here we can apply LangSec principles I will discuss later lecture
- Split application up in multiple components/processes, to then
 - run some with lower privileges
 - compartmentalisation or sandboxing
 - implement some in memory safe programming language?

Organisational aspects

Getting organisation & processes in place to make sure that all this happens!

- **adopt some process for secure development**
(aka secure SDLC, e.g. Microsoft SDL or something home-grown)
 - recall first lecture
 - as required by CRA
- **embed measures** from previous slides (and rest of this course) **in that process**
 - Risk analysis will be a first step, which helps to identify high risk components

Not forgetting the human factor!

- **Training**
Common good practice: set up a **Security Champion program**

Static Analysis

Static analysis aka source code analysis aka SAST

Automated analysis *at compile time* to find *potential bugs*

Broad range of techniques, from light- to heavyweight:

1. simple **syntactic checks**, incl. `grep` or `CTRL-F`

```
grep " gets(" *.cpp
```

2. **type checking**

eg. warning if an `int` is added to a `bool`

3. **more advanced analyses** taking **semantics** into account using:
data flow analysis, control flow analysis, abstract interpretation,
symbolic evaluation, constraint solving, program verification, model
checking, ...

All compilers do some static analysis

Lightweight static analysis tools also called **source code scanners**

Tools aiming at security: **SAST**

False positives & false negatives

Important quality measures for any static analysis:

A. rate of **false positives**

- tool complains about non-error

B. rate of **false negatives**

- tool fails to complain about error

Which do you think is worse?

False positives are worse, as they kill usability !!

Alternative (confusing!) terminology: analysis can be called

- sound it only finds *real* bugs, ie. no false positives
- complete it finds *all* bugs, ie. no false negatives

Very simple static analysis: *style checking*

- Warning about **bad names & violations of conventions**, eg
 - constants not written in ALL CAPS
 - Java method starting with capital letter
 - C# method starting with lower case letter
 - ...
- Enforcing other (company/project-specific) naming conventions and coding guidelines

More interesting static analyses

- Warning about **unused variables**
- Warning about **dead/unreachable code**
- Warning about **missing initialisation**
 - possibly as part of language definition (eg in Java) and checked by compiler

This may involve

control flow analysis

`if (b) { c = 5; } else { c = 6; }` initialises c

`if (b) { c = 5; } else { d = 6; }` does not

data flow analysis

`d = 5; c = d;` initialises c

`c = d; d = 5;` does not

Spot the defect!

```
BOOL AddTail(LPVOID p) {  
    ...  
    if (queue.GetSize() >= this->_limit);  
    {  
        while(queue.GetSize() > this->_limit-1)  
        {  
            ::WaitForSingleObject(handles[SemaphoreIndex],1);  
            queue.Delete(0);  
        }  
    }  
}
```



Suspicious code in xpdfwin found by PVS-Studio (www.viva64.com).

V529 Odd semicolon ';' after 'if' operator.

Note that this is a very simple syntactic check!

You could (should?) use coding guidelines that disallow this, even though it is legal C++

Spot the security flaw! CVE-2014-1266

```
static OSStatus SSLVerifySignedServerKeyExchange (SSLContext *ctx, bool isRsa, SSLBuffer
signedParams, uint8_t *signature, UInt16 signatureLen)
{ OSStatus err;
  ..
  if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
  if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
  goto fail; 
  if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
  ...
fail:
  SSLFreeBuffer(&signedHashes);
  SSLFreeBuffer(&hashCtx);
}
```

Infamous goto bug in Apple iOS implementation of TLS

- **Dead code analysis** would reveal this flaw!
- Or a code style that insists you **always use { }** for branches

Spot the defects!

```
void start_engine_control() {  
    char* buf2 = malloc (2*SOME_CONSTANT);  
    char* buf = malloc (SOME_CONSTANT);  
    start_engine();  
    memset(buf2, 0, SOME_CONSTANT);  
    // initialise first half of buf2 to 0  
  
    // main loop  
    while (true) {  
        get_readings(buf,buf2);  
        perform_engine_control(buf,buf2);  
    }  
}
```

Possible integer overflow

Can be hard to check statically, but for constant is should be doable

No check if mallocs succeeded!!
(easier to check syntactically)

Check you mallocs!

```
void start_engine_control() {  
    ...  
    char*  buf = malloc (SOME_CONSTANT);  
    if (buf == NULL) { // now what?!?!?  
        exit(0);      // or something more graceful??  
    }  
    ...  
    start_engine();  
    ...  
    perform_engine_control(buf);  
}
```

Typically, the place where malloc fails is the place to think about what to do

Spot the defect :-)



First Ariane V launch

integer overflow in conversion of 64 bit float to 16 bit int

https://www.youtube.com/watch?v=PK_yguLapgA

Limits of static analysis

Does

```
if (i < 5 ) { c = 5; }  
if ((i < 0) || (i*i > 20 )) { c = 6; }
```

initialise c?

Many analyses become hard – or undecidable - at some stage

Analysis tools can then:

- report that they “DON'T KNOW”
- give a (possible) false positive
- give a (possible) false negative

Example source code analysis tools

- free tools for Java: CheckStyle, PMD, SpotBugs (formerly FindBugs)
- for C(++) from Microsoft: PREfast, FxCop
- outdated, but free tools focusing on security
 ITS4 and Flawfinder (C, C++), RATS (also Perl, PHP)
- commercial
 Coverity (C/C++) , PolySpace (C/C++, Ada), SparkAda (Ada),
 Fortify, Klocwork, PVS-Studio, IBM AppScan, VeraCode,
 CheckMarx, SonarQube,
 CodeQL (aka Semmle), semgrep

easy & fun
to download
and try out!

Some tools focus on C/C++, others on web applications

Such tools can be useful, but... **a fool with a tool is still a fool**

Safer C

Safer string libraries in C

- `strcpy(dst,src)` , `strcat(dst,src)`

Error prone! Memory corruption if strings are not null-terminated or too long

- `strncpy(dst,src,n)` , `strncat(dst,src,n)`

Better: takes **max # of bytes to be copied** as parameter

But: can result in non-null-terminated strings if strings don't fit ☹

- `strncpy(dst,src,dst_len)` , `strncat(dst,src,dest_len)`

Result is guaranteed to be null-terminated string

Size of destination array as parameter, not # of chars to be copied,
which gives simpler code when e.g. concatenating multiple strings

Another alternative: here `strncpy` and `strncat`

Date Thu, 18 Jun 2026 19:49:50 -0700
From Kees Cook <>
Subject [GIT PULL] strncpy-removal updates for v7.2-rc1

Hi Linus,

Please pull the strncpy removal for v7.2-rc1. BPF's last usage removal has landed, so here we go.

Thanks!

-Kees

Linux Finally Killed strncpy. It Took Six Years, 362 Patches, and 70 Contributors.

The story of a 40-year-old C function that looked safe but wasn't, and the codebase-wide surgery that removed it from the kernel entirely.

STRNCPY() — REMOVED IN LINUX 7.2

CASE 1: SOURCE FILLS BUFFER (SRC ≥ N)



△ No null terminator written. Code reading this buffer overruns into adjacent kernel memory — credentials, keys, pointers.

CASE 2: SOURCE SHORTER THAN BUFFER



△ 5 redundant zero-writes after null. No benefit. Pure CPU waste at scale.

STRSCPY() — REPLACEMENT (PRIMARY)

CASE 1: SOURCE FILLS BUFFER (SRC ≥ N)



✓ Always null-terminated. Returns -E2BIG on truncation — caller knows immediately, no silent overflow.

CASE 2: SOURCE SHORTER THAN BUFFER



✓ Stops after null. Remaining bytes untouched. No redundant writes.

Character data Null terminator \0 (safe) Missing terminator (danger) Wasteful zero-fill

Untouched (free)

To read: <https://chessman7.substack.com/p/linux-finally-killed-strncpy-it-took>

More type information: Nullability qualifiers

LLVM/Clang introduced type qualifiers

`__nullable`

`__nonnull`

for more precise typing & hence better static analysis for bugs

- Compiler can then warn about missing null-checks

Example: `char *strchr(char *str, int c)`, which finds the first occurrence of `(char)c` in null-terminated string `str`, becomes

```
__nullable char *strchr(__nonnull char *s, int c)
```

gcc has a clumsier `nonnull` attribute

```
void my_memcpy (__attribute__((nonnull)) char *dest,  
                __attribute__((nonnull)) char *src, int len)  
    __attribute__((returns_nonnull))
```

Spot the (potential) bugs!

```
void zero(int a, size_t len) {
    for (int i = 0; i <= len; i++) { a[i] = 0; }
}

-----

typedef struct {
    int* buf;
    size_t count;
} sized_buf;

void zero_sb(sized_buf* ptr) {
    sized_buf sb = *ptr; // the sized_buf struct that ptr points to
    for (int i = 0; i <= sb.count ; i++) { sb.buf[i] = 0; }
} // for C programmers: sb.count could be written as ptr->count

-----

int main() {
    int[10] a; int* b; ... b = a++; ...
    sized_buf* ptr; ... ptr++; ... zero_sb(ptr); ...
}
```

Spot the (potential) bugs!

```
void zero(int a, size_t len) {
    for (int i = 0; i <= len; i++) { a[i] = 0; }
} // ^^ n is probably the length of a, so <= should be <
-----

typedef struct {
    int* buf;
    size_t count;
} sized_buf;

void zero_sb(sized_buf* ptr) {
    sized_buf sb = *ptr; // the sized_buf struct that ptr points to
    for (int i = 0; i <= sb.count ; i++) { sb.buf[i] = 0; }
} // ^^ count is probably the length of buf, so <= should be <
-----

int main() {
    int[10] a; int* b; ... b = a++ ; .. // a++ makes sense because a is array
    sized_buf* ptr; ... ptr++; ... zero_sb(ptr); ...
} // ^^ makes no sense if ptr does not point to array
```

-fbounds-safety annotations

```
void zero(int *__counted_by(len) a, size_t len) {  
    for (int i = 0; i <= len; ++i) {a[i] = 0;}  
    // Clear out of bounds access now  
}
```

```
typedef struct {  
    int *__counted_by(count) buf;  
    size_t count;  
} sized_buf;
```

```
sized_buf *__single ptr;  
    // For pointers to single (non-array) objects  
    // So p++ or p[2] will result in error
```

These annotations 1) let programmers express their intent, 2) can improve static analysis, and 3) can be used to insert compile-time checks

-fbounds-safety in LLVM/clang (since 2024)

- C compiler inserts bound checks made possible by annotations notably `__counted_by( )`
 - Compiler can infer some annotations automatically.
 - Compiler can be made to complain about missing annotations
- Checks are meant to be left in production code
 - Runtime overhead & developer effort to annotate code acceptable
- Binary compatible aka preserves ABI (Application Binary Interface)
 - so usage can be introduced gradually
- Supported for C and Swift, not (yet?) Objective C and C++
- Developed by & heavily used at Apple

Not exam material but more info at

EuroLLVM 2025 talk 'Adopting -fbounds-safety in practice'

<https://www.youtube.com/watch?v=rlevLn831R4>

-fbounds-safety annotations

Some variants of `__counted_by`

- `__counted_by_or_null(...)`
for arrays that can be null; compiler will add null-checks
- `__sized_by(...)` `__sized_by_or_null(...)`
to express size in bytes instead of elements

Eg

```
void* __sized_by_or_null(len) malloc(size_t len);
```

__null_terminated strings

-fbounds-safety also has an annotation for null-terminated strings

```
char *__null_terminated s;
```

```
void process_string(char *__null_terminated str);
```

Note that `char*` is overloaded: to pass arrays as parameters to functions, we have to use `char*`

```
char[8] udp_header1;           // UDP header on stack
```

```
char* str = "Welcome";        // 8-byte string
```

```
process_header(udp_header);
```

```
void process_header(char* x) {  
    // x could be an array or a string  
    ... }
```

This is an unavoidable ambiguity in C.

-fbounds-safety *inferred* annotations

Often annotations can be left implicit

```
void f(int n) {  
    int[12] a;    // Compiler infers __counted_by(12)  
    a[n] = 1234; // Out-of-bounds access will result  
                // in runtime-error  
}
```

-fbounds-safety wide pointers

Wide pointer with upper & lower bound,
instead of a length field as discussed for fat pointers last week

```
typedef struct {  
    int *pointer;  
    int *upper_bound;  
    int *lower_bound;  
} wide_pt;
```

Clang compiler inserts bound checks on use of wide pointers

Obvious downside: not binary compatible

(weaker) gcc alternative: `-D_FORTIFY_SOURCE`

Compiler flag to insert some lightweight defensive checks for some memory and string operations.

Introduced in gcc in 2004

Version `-D_FORTIFY_SOURCE=3` since 2022

Microsoft alternative: SAL

Microsoft SAL annotation language, not for *dynamic instrumentation*,
but for *static analysis* with PREfast tool

```
void zero(_Out_writes_(len) int* a, int len) {  
    for (int i = 0; i < len; ++i) { a[i] = 0; }  
}  
  
_checkReturn _Ret_opt_bytecount_(len) void* malloc(size_t len);  
  
// _opt_ signals possible nullness  
// _checkReturn forces caller to check return value for nullness
```

PREfast does static analysis for spatial memory vulnerabilities, incl. reading of uninitialised memory, and null-dereferences, plus some ad-hoc checks

SAL annotations improve precision & enable analysis per procedure

PREfast is included in Visual Studio as `/analyze` flag since 2005

Safer C (or C++): Custom allocators

Custom implementations of malloc & free can improve security

e.g. by

- keeping deallocated memory in **quarantine** for a while
- **randomising** freelist
- using **different partitions for different types** to prevent type-confusion bugs
- using **memory tagging**, esp. ARM MTE
- surrounding memory chunks with **guard pages**

Examples:

- Google PartitionAlloc used in Chrome
- Scudo Hardened Allocator used in Android
- IsoHeap GigaCage in WebKit
- AUTOSlab by grsecurity
- Apple's iBoot

Safer C++

array and vector in C++

Instead of C strings `char*` and C arrays `int[]`,
C++ has safer alternatives:

```
std::string s;  
std::array<int,N> a;  
std::vector<int> v;
```

Not 100% memory safe, but a lot memory-*safer*,
esp. against spatial bugs

- `a[i]` is still not bounds-checked, but `a.at(i)` is
- `data()` gives raw pointer access to internal data representation
- `std::array` preserves its type & length when passed to function
Note that in C an `int[]` can only be passed as argument as an `int*` pointer
- Built-in function for copying and iteration (without having to worry about lengths)
But iterators can easily become dangling pointers

Next week:
smart pointers in C++
to help prevent temporal bugs