What's Hot

• 1970s
  - Structured Programming
• 1980s
  - User Friendly
• 1990s
  - Object Oriented Programming
• 2000s
  - Generic/Meta Programming
And Now...

SECURITY
21  Hacked Cameras, DVRs Powered Today’s Massive Internet Outage

A massive and sustained Internet attack that has caused outages and network congestion today for a large number of Web sites was launched with the help of hacked “Internet of Things” (IoT) devices, such as CCTV video cameras and digital video recorders, new data suggests.

Earlier today cyber criminals began training their attack cannons on Dyn, an Internet infrastructure company that provides critical technology services to some of the Internet’s top destinations. The attack began creating problems for Internet users reaching an array of sites, including Twitter, Amazon, Tumblr, Reddit, Spotify and Netflix.

A depiction of the outages caused by today’s attacks on Dyn, an Internet infrastructure company. Source: DownDetector.com.

At first, it was unclear who or what was behind the attack on Dyn. But over the past few hours, at least one computer security firm has come out saying the attack involved Mirai, the same malware strain that was used in the record 620 Gbps attack on my site last month. At the end September 2016, the hacker responsible for creating the Mirai malware released the source code for it, effectively letting anyone build their own attack army using Mirai.

Mirai severs the Web for IoT devices protected by little more than factory-default usernames and passwords, and then enlists the devices in attacks that hurl junk traffic at an online target until it can no longer accommodate legitimate visitors or users.
Prediction

Memory safety will become a requirement for programming languages
Memory Safety

“a concern in software development that aims to avoid software bugs that cause security vulnerabilities dealing with random-access memory (RAM) access, such as buffer overflows and dangling pointers”

https://en.wikipedia.org/wiki/Memory_Safety
a.k.a.

Pointers Gone Wild!
The Usual Suspects

- Buffer overflow
- Pointer arithmetic
- Uninitialized pointers
- Casting
- Misaligned pointers
- Pointers to expired stack frames
- Dangling pointers
int[10] a;
for (size_t i = 0; i <= 10; ++i)
a[i] = ...;
Solution

• arrays do not decay to pointers
  – array dimension is carried with it
• runtime array bounds checking
Arrays Look Like

```
struct Array {
    size_t length;
    T* ptr;
}
```
Pointer Arithmetic

int[10] a;
int* p = &a[0];
for (size_t i = 0; i <= 10; ++i)
    p[i] = ...;

Lost the array bounds checking
Solution

Pointer arithmetic is not allowed
Uninitialized Pointers

```c
int p;
...
*p = ...;
```
Solution

Default initialize to null
Uninitialized Pointers 2

```c
struct S {
    int i;
    char* p;
    this(int x) { this.i = x; }
}

S s = S(3);
*s.p = ...;  // error, p not initialized
```
Same Solution

Default initialize the fields
No Initialization

S s = void;

(But not allowed in @safe code)
Casting

int* p = cast(int*) 1234;  // error
int i = cast(int) p;        // ok
Unions

union U
{
    int* p;
    int i;
}

U u;
u.i = 1234;
int* p = u.p; // error
Size Casts

```c
char* pc;
byte* pb = cast(byte*) pc; // ok
int* pi = cast(int*) pc;    // nope
```
Misaligned Pointers

struct S
{
    align (1):
    byte b;
    int i;
}

S s;
int* p = &s.i;  // error
int* foo()
{
    int i;
    return &i;  // error
}
Easily Defeated

```c
int* foo()
{
    int i;
    int* p = &i;
    return p;
}
```
Using `ref` Instead

- `ref` is a restricted pointer
  - can only use `ref` for parameters/returns
  - no arithmetic
  - no escape
ref int foo()
{
    int i;
    ref int p = i; // not allowed!
    return p;
}
But Need Identity Function

```c
ref int foo(ref int i)
{
    return i;
}
```
Leading To

ref int foo(ref int i)
{
    return i;
}

ref int bar()
{
    int i;
    return foo(i);  // uh-oh!
}
Introducing `return ref`

```c
ref int foo(int i)
{
    return i;
}

ref int bar()
{
    int i;
    int j = foo(i);  // ok
    return foo(i);  // error
}
```
Back To Pointers

int* foo(int* i)
{
    return i;
}

int* bar()
{
    int i;
    return foo(&i); // boom!
}
Introducing **scope**

void abc(int*);
int* q;

int* foo(scope int* p, int** pp)
{
    abc(p);    // error
    q = p;      // nope
    *pp = p;   // nice try
    return p; // no way
}
Locals Can Be `scope`

```c
void foo(scope int* p)
{
    scope int* q = p; // ok
    int* s = p; // ok, `scope` is inferred
}
```
int j;

scope int* p;
int i;

p = &j;  // ok
p = &i;  // error
p = c ? &i : &j;  // error
More Inference

```c
void foo( ) (int* p)
{
    *p = 3;
    // 'p' is inferred as `scope`
}
```

Lambdas too
int* foo(int* p) {
    return p;
}

int* bar(int* q) {
    int i;
    *foo(&i) = 3; // ok
    return foo(q); // ok
    return foo(&i); // error
}
What About

- dynamic arrays
  - ptr / length pair
- delegates
  - ptr / funcptr pair
- this
  - ptr or ref
Dangling Pointers

```c
int* p = cast(int*) malloc(5 * int.sizeof);
...
free(p);
*p = 3;  // Boom!
```
“Oh I don't think there's anything in that black bag for me.” – Dorothy
Containers

- Use RAII (scoped destruction)
- prevent general escape of pointers
- use return scope and return ref

But still some work to be done
Conclusion

- wild pointers can be corralled
- simple annotations are possible
- many (most?) annotations can be inferred
- system code still allowed
What if I were the worst comic hero ever?

lame

NOT FUN

You'll get used to it.

http://dlangcomicstrips.tumblr.com