School of Computing Science

60 YEARS OF COMPUTING AT GLASGOW

Writing Secure Code

Networked Systems (H)

Lecture 18

Lecture Outline

- Developing secure network applications:
  - The robustness principle
  - Validating input data
  - Writing secure code:
    - Example: classic buffer overflow attack
    - Arbitrary code execution
  - Discussion

The Robustness Principle (Postel's Law)

> At every layer of the protocols, there is a general rule whose application can lead to enormous benefits in robustness and interoperability:
>
> "Be liberal in what you accept, and conservative in what you send"
>
> Software should be written to deal with every conceivable error, no matter how unlikely; sooner or later a packet will come in with that particular combination of errors and attributes, and unless the software is prepared, chaos can ensue. In general, it is best to assume that the network is filled with malevolent entities that will send in packets designed to have the worst possible effect. This assumption will lead to suitable protective design, although the most serious problems in the Internet have been caused by un-envisaged mechanisms triggered by low-probability events; mere human malice would never have taken so devious a course!
>
> RFC1122

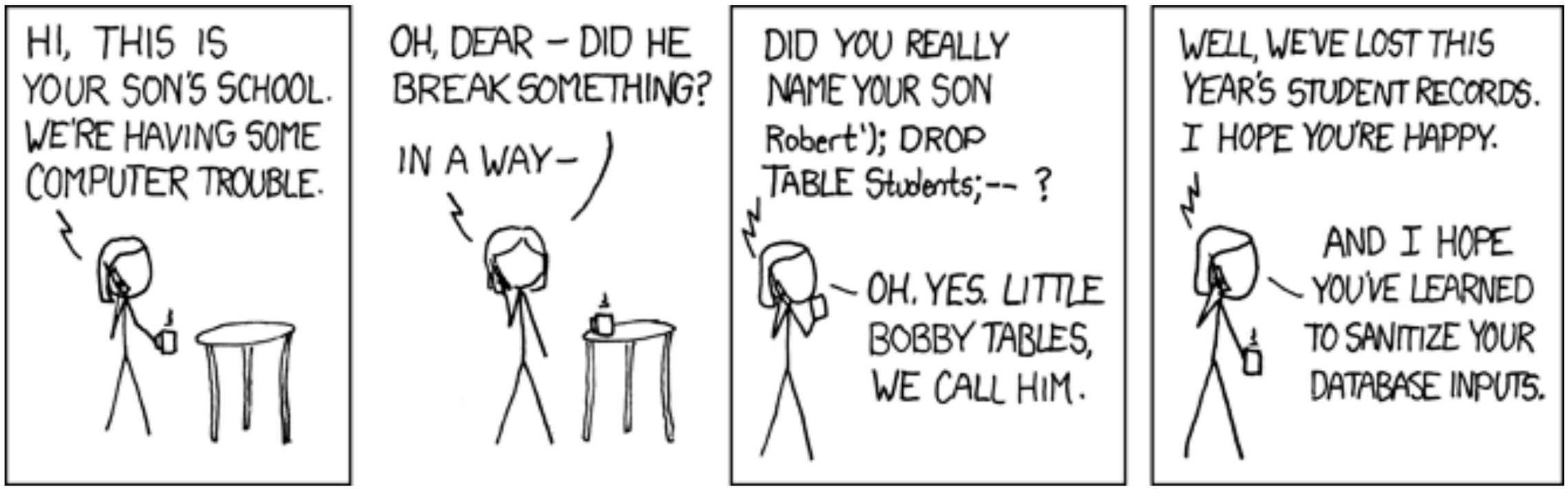
- Balance interoperability with security – don't be *too* liberal in what you accept; a clear specification of how and when you will fail might be more appropriate

The Robustness Principle (Postel's Law)

"Postel lived on a network with all his friends.
We live on a network with all our enemies.
Postel was wrong for todays internet."

— *Poul-Henning Kamp*

Validating Input Data



http://xkcd.com/327/

Validating Input Data

- Networked applications fundamentally dealing with data supplied by un-trusted third parties
  - Data read from the network may not conform to the protocol specification
  - Due to ignorance and/or bugs
  - Due to malice, and a desire to disrupt services
- Must carefully validate all data before use

Writing Secure Code

- The network is hostile: any networked application is security critical
  - Must carefully specify behaviour with both correct and incorrect inputs
  - Must carefully validate inputs and handle errors
  - Must take additional care if using type- and memory-unsafe languages, such as C and C++, since these have additional failure modes

Example: Classic Buffer Overflow Attack

- Memory-safe programming languages check array bounds
  - Fail cleanly with exception on out-of-bound access
  - Behaviour is clearly defined at all times
- Unsafe languages, such as C and C++, don't check
  - Responsibility of the programmer to ensure bounds are not violated
  - Easy to get wrong – typically results in a "core dump" – or undefined behaviour
  - What actually happens here?

Function Calls and the Stack

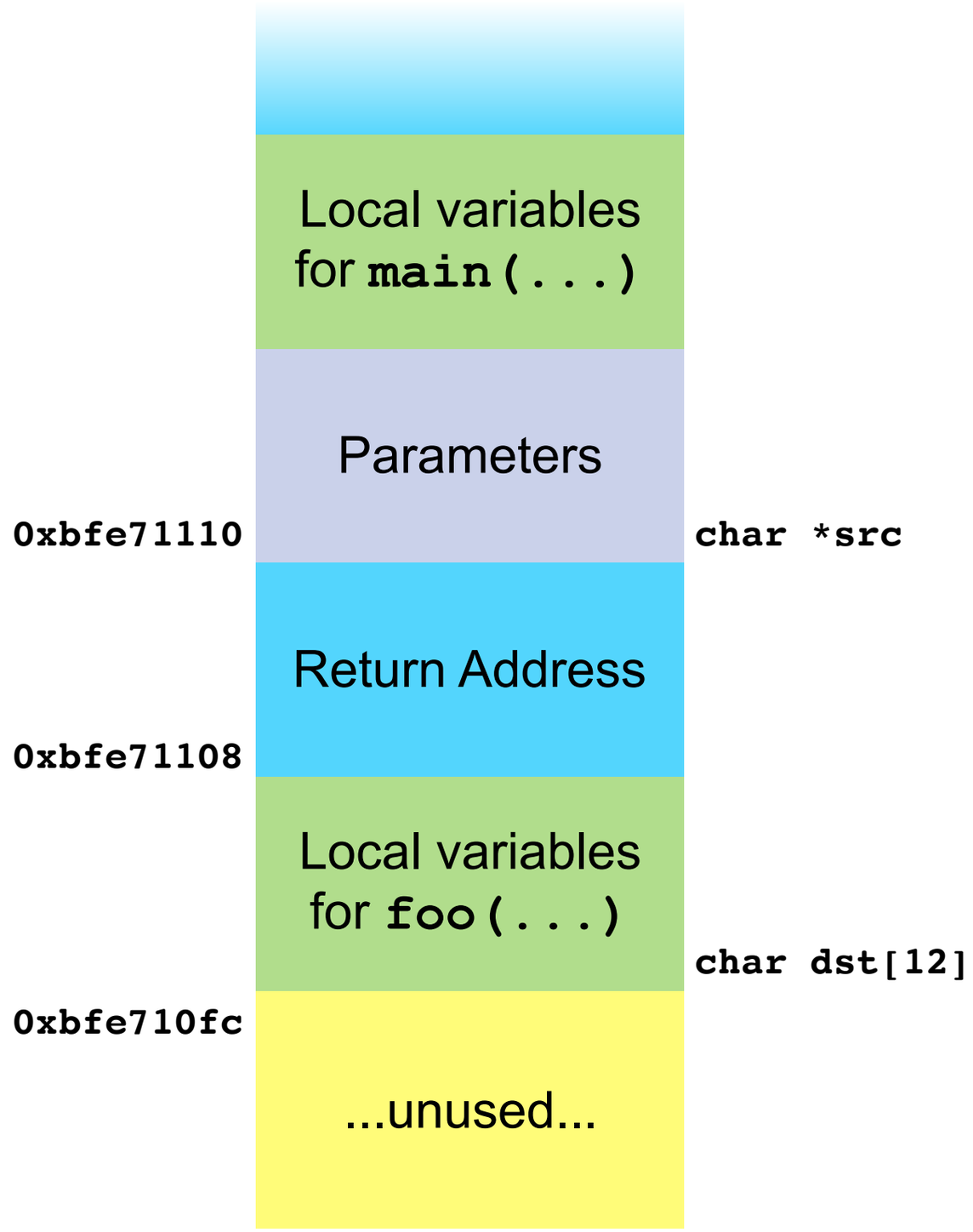
```
// overflow.c
#include <string.h>
#include <stdio.h>

static void
foo(char *src)
{
        char dst[12];

        strcpy(dst, src);
}

int
main(int argc, char *argv[])
{
        char hello[] = "Hello, world\n";

        foo(argv[1]);
        printf("%s", hello);
        return 0;
}
```

```
$ gcc overflow.c  -o overflow
$ ./overflow 123456789012
Hello, world
$ ./overflow 1234567890123
Abort trap (core dumped)
$
```

What happens when `argv[1]` is longer than 12 bytes?

Function Calls and the Stack

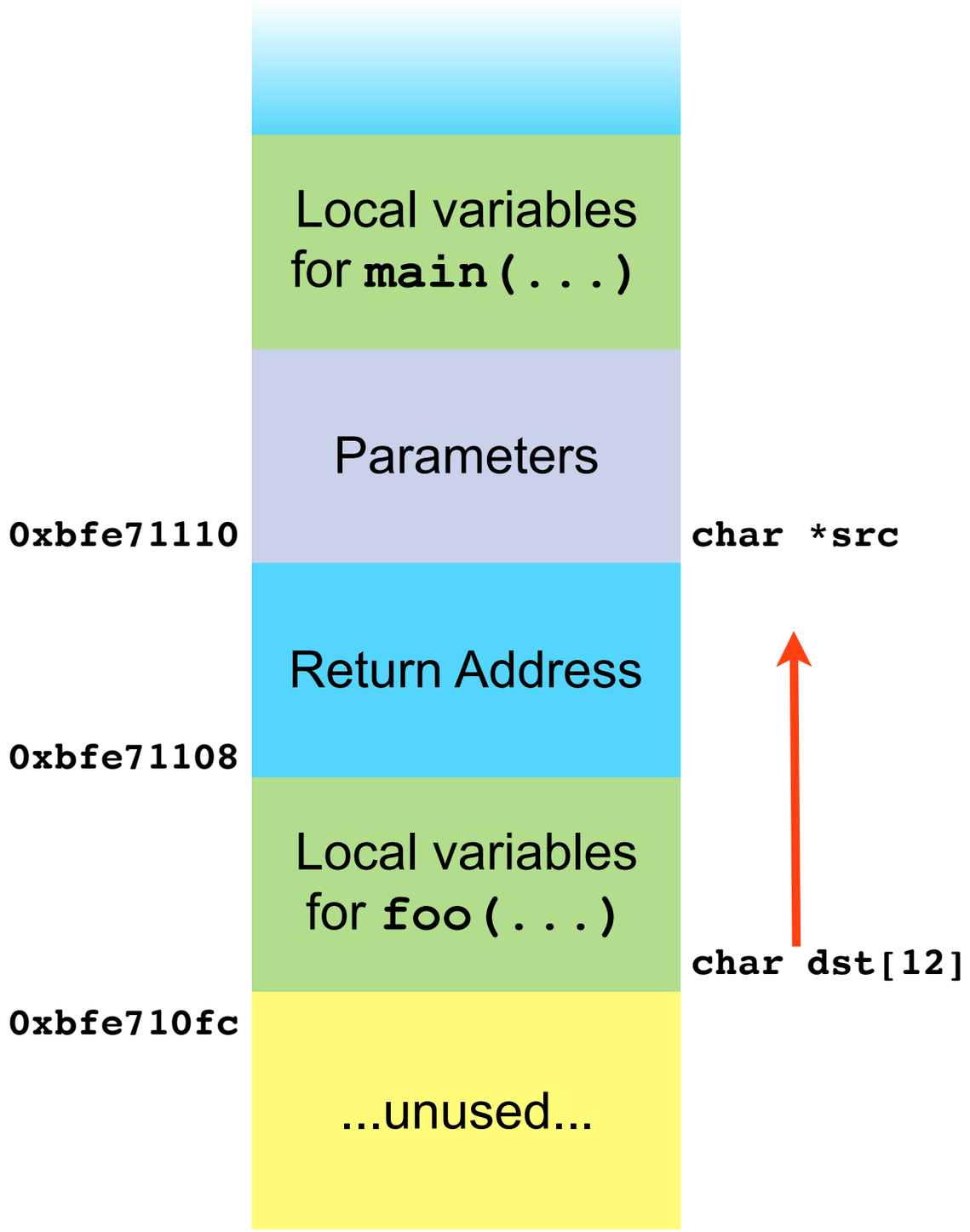
```
// overflow.c
#include <string.h>
#include <stdio.h>

static void
foo(char *src)
{
        char dst[12];

        strcpy(dst, src);
}

int
main(int argc, char *argv[])
{
        char hello[] = "Hello, world\n";

        foo(argv[1]);
        printf("%s", hello);
        return 0;
}
```



Example of call stack within the call to the function `foo()`

Function Calls and the Stack

- The `strcpy()` call doesn't check array bounds
- Overwrites the function return address on stack, along with the following memory locations
- If malicious, we can write executable code into this space, set return address to jump into our code…



Example of call stack within the call to the function `foo()`

Arbitrary Code Execution

- Buffer overflows in network code are one of the main sources of security problems
  - If you write network code in C/C++, be very careful to check array bounds
  - If your code can be crashed by received network traffic, it probably has an exploitable buffer overflow
  - http://insecure.org/stf/smashstack.html

Discussion

- Many networked applications written in memory- or type-unsafe languages
  - Many good historical reasons for this, and clearly will take time to replace old deployments with safe alternatives
  - Is it justifiable to write new networked code in this way, now that there are safe alternatives?
    - Java, C#, Swift, Rust, …
- As engineers, we have a duty to use best practices – could you defend your implementation choices?