University of Glasgow

School of Computing Science

Security Considerations

Networked Systems (H)

Lecture 9

Lecture Outline

- Pervasive traffic monitoring
- Confidentiality and authentication
- Securing network transport
- Writing secure code

Network Monitoring and the Need for Encryption

Network Monitoring and the Need for Encryption

- Possible to intercept traffic on a network
- Many countries monitor traffic for legal reasons
  - Much is desirable – good reasons for law enforcement to intercept *some* traffic – but Edward Snowden showed pervasive monitoring widespread
  - IETF consensus: “we cannot defend against the most nefarious actors while allowing monitoring by other actors no matter how benevolent some might consider them to be, since the actions required of the attacker are indistinguishable from other attacks”

    [RFC 7258 “Pervasive Monitoring is an Attack” – https://tools.ietf.org/html/rfc7258]

Edward Snowden

- Organisations may monitor traffic for business reasons
  - “Your call may be monitored for quality and training purposes” – regulatory requirements to be able to monitor some traffic
  - To support network operations and trouble-shooting

[Moriarty and Morton, “Effect of Pervasive Encryption on Operators”, https://tools.ietf.org/html/draft-mm-wg-effect-encrypt]

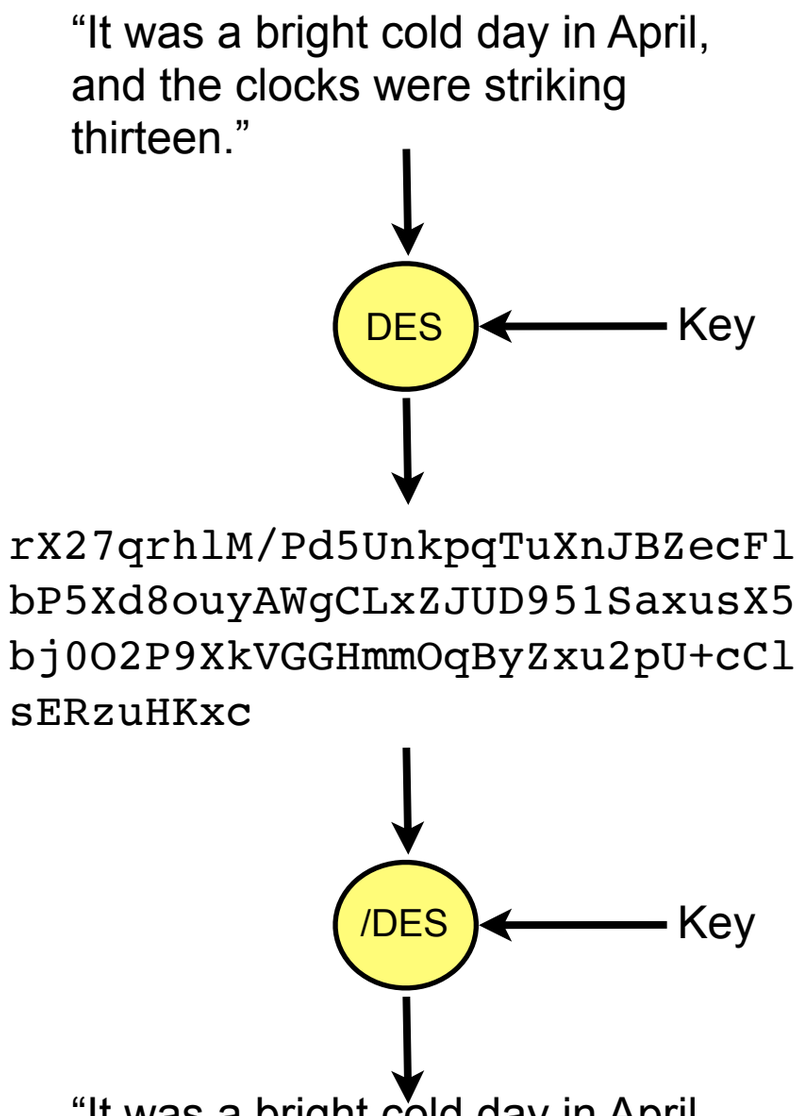
- Malicious users may monitor traffic on a link
  - For example, many Wi-Fi links have poor security allowing anyone on the same Wi-Fi network to observe all traffic on that network
  - Hacked routers may allow monitoring of backbone links
  - Steal data and user credentials; identity theft; active attacks

Confidentiality

- Must encrypt data to achieve confidentiality
- Two basic approaches
  - Symmetric cryptography
    - Advanced Encryption Standard (AES)
  - Public key cryptography
    - The Diffie-Hellman algorithm
    - The Rivest-Shamir-Adleman (RSA) algorithm
    - Elliptic curve-based algorithms
- Complex mathematics – will not attempt to describe

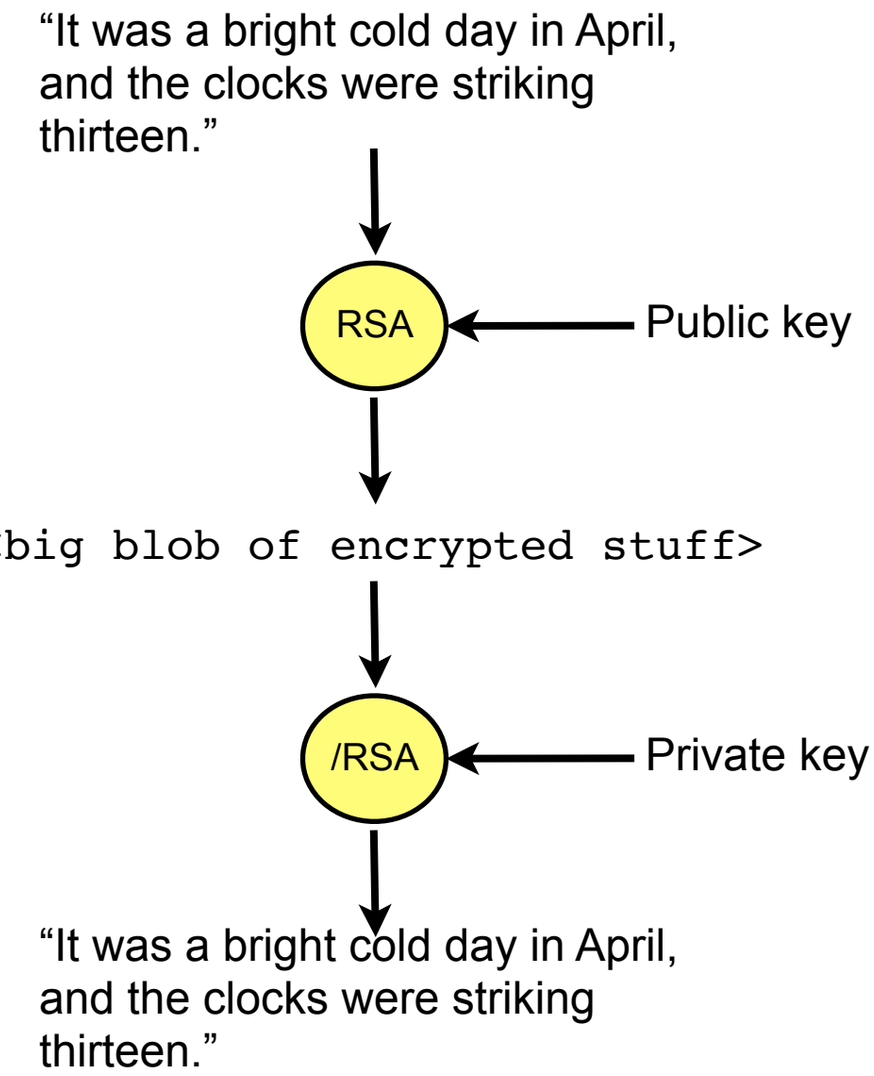
Symmetric Cryptography

- Function converts plain text into cipher-text
  - Fast – suitable for bulk encryption
  - Cipher-text is binary data, and may need base64 encoding
- Conversation is protected by a secret key
  - The same key is used to encrypt as is used to decrypt
  - Key must be kept secret, else security lost – a problem: how to distribute the key?



Public Key Cryptography

- Key split into two parts:
  - Public key – is widely distributed
  - Private key – must be kept secret
- Encrypt using public key → need private key to decrypt
  - Public keys are published in a well known directory → solves the key distribution problem
  - Problem: very slow to encrypt and decrypt



Hybrid Cryptography

- Use combination of public-key and symmetric cryptography for security and performance
  - Generate a random, ephemeral, *session key* that can be used with symmetric cryptography
  - Use a public-key system to securely distribute this session key – relatively fast, since session key is small
  - Encrypt the data using symmetric cryptography, keyed by the session key
- Example: Transport Layer Security (TLS) protocol used with HTTP

Authentication

- Encryption can ensure confidentiality – but how to tell if a message has been tampered with?
  - Use combination of a *cryptographic hash* and public key cryptography to produce a *digital signature*
  - Gives some confidence that there is no *man-in-the-middle* attack in progress
- Can also be used to prove origin of data

Cryptographic Hash Functions

- Generate a fixed length (e.g., 256 bit) hash code of an arbitrary length input value
  - Should not be feasible to derive input value from hash
  - Should not be feasible to generate a message with the same hash as another
- Examples:
  - MD5 and SHA-1 (both are broken – do not use)
  - SHA-2 (a.k.a., SHA-256)

SHA256(“It was a bright cold day in April, and the clocks were striking thirteen”)
= 0fc5c1f4082e697b211cdfa12479b4b3dd57c8da69c8904f5e0fc32499cf4245

Digital Signature Algorithms

- Generating a digital signature:
  - Generate a cryptographic hash of the data
  - Encrypt the hash with your *private key* to give a *digital signature*
- Verifying a digital signature:
  - Re-calculate the cryptographic hash of the data
  - Decrypt the signature using the public key, compare with the calculated hash value → should match

Existing Secure Protocols

- Existing security protocols give confidentiality and authentication:
  - IPSec – useful for VPNs
  - Secure Sockets Layer (SSL) – obsolete and broken, use TLS instead
  - Transport Layer Security (TLS v1.2 *or later*) – general purpose security for TCP-based applications
  - Datagram TLS – for securing UDP-based applications
  - Secure RTP – for securing interactive multimedia applications
  - Secure shell (ssh) – for securing remote login applications
- Use them – don't try to invent your own!

Using TLS

- IETF provides guidelines for how best to use TLS: https://tools.ietf.org/html/rfc7525
  - Read this if you use TLS in your application – and check for updates first
  - IETF “Using TLS in Applications” working group https://datatracker.ietf.org/wg/uta/charter/

- State-of-the-art in TLS implementations is in flux
  - OpenSSL is popular, but poor quality
    - Alternatives in rapid development – not clear which is the best long term option
  - For macOS or Windows, use the system libraries

Key Escrow

Keys Under Doormats:

MANDATING INSECURITY BY REQUIRING GOVERNMENT ACCESS TO ALL DATA AND COMMUNICATIONS

Harold Abelson, Ross Anderson, Steven M. Bellovin, Josh Benaloh, Matthew Blaze, Whitfield Diffie, John Gilmore, Matthew Green, Peter G. Neumann, Susan Landau, Ronald L. Rivest, Jeffrey I. Schiller, Bruce Schneier, Michael Specter, Daniel J. Weitzner

Abstract

Twenty years ago, law enforcement organizations lobbied to require data and communication services to engineer their products to guarantee law enforcement access to all data. After lengthy debate and vigorous predictions of enforcement channels "going dark," these attempts to regulate the emerging Internet were abandoned. In the intervening years, innovation on the Internet flourished, and law enforcement agencies found new and more effective means of accessing vastly larger quantities of data. Today we are again hearing calls for regulation to mandate the provision of exceptional access mechanisms. In this report, a group of computer scientists and security experts, many of whom participated in a 1997 study of these same topics, has convened to explore the likely effects of imposing extraordinary access mandates.

We have found that the damage that could be caused by law enforcement exceptional access requirements would be even greater today than it would have been 20 years ago. In the wake of the growing economic and social cost of the fundamental insecurity of today's Internet environment, any proposals that alter the security dynamics online should be approached with caution. Exceptional access would force Internet system developers to reverse "forward secrecy" design practices that seek to minimize the impact on user privacy when systems are breached. The complexity of today's Internet environment, with millions of apps and globally connected services, means that new law enforcement requirements are likely to introduce unanticipated, hard to detect security flaws. Beyond these and other technical vulnerabilities, the prospect of globally deployed exceptional access systems raises difficult problems about how such an environment would be governed and how to ensure that such systems would respect human rights and the rule of law.

- Effective security is difficult – failures tend to be due to bad implementations or protocols, not weak crypto
- So-called exceptional access or key escrow systems will be discovered, and exploited, by malicious actors – we do not have the expertise to secure such systems
- Design to limit access to keying material

H. Abelson, *et al.*, "Keys under doormats: Mandating insecurity by requiring government access to all data and communications", MIT Computer Science and Artificial Intelligence Lab, technical report MIT-CSAIL-TR-2015-026, July 2015. http://dspace.mit.edu/handle/1721.1/97690

Summary

- Pervasive monitoring
- Confidentiality, authentication, and crypto
- Secure transport protocols
- Key escrow

Writing Secure Networked Applications

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)

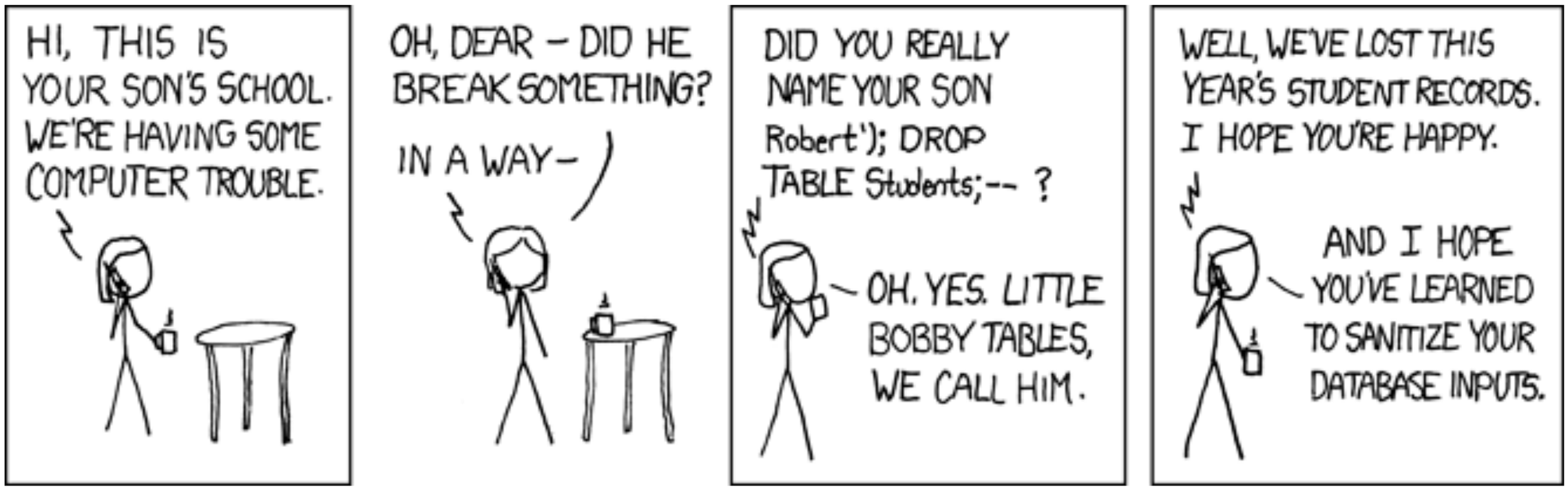
"Be liberal in what you accept, and
 conservative in what you send"

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*

See also: https://datatracker.ietf.org/doc/draft-thomson-postel-was-wrong/

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

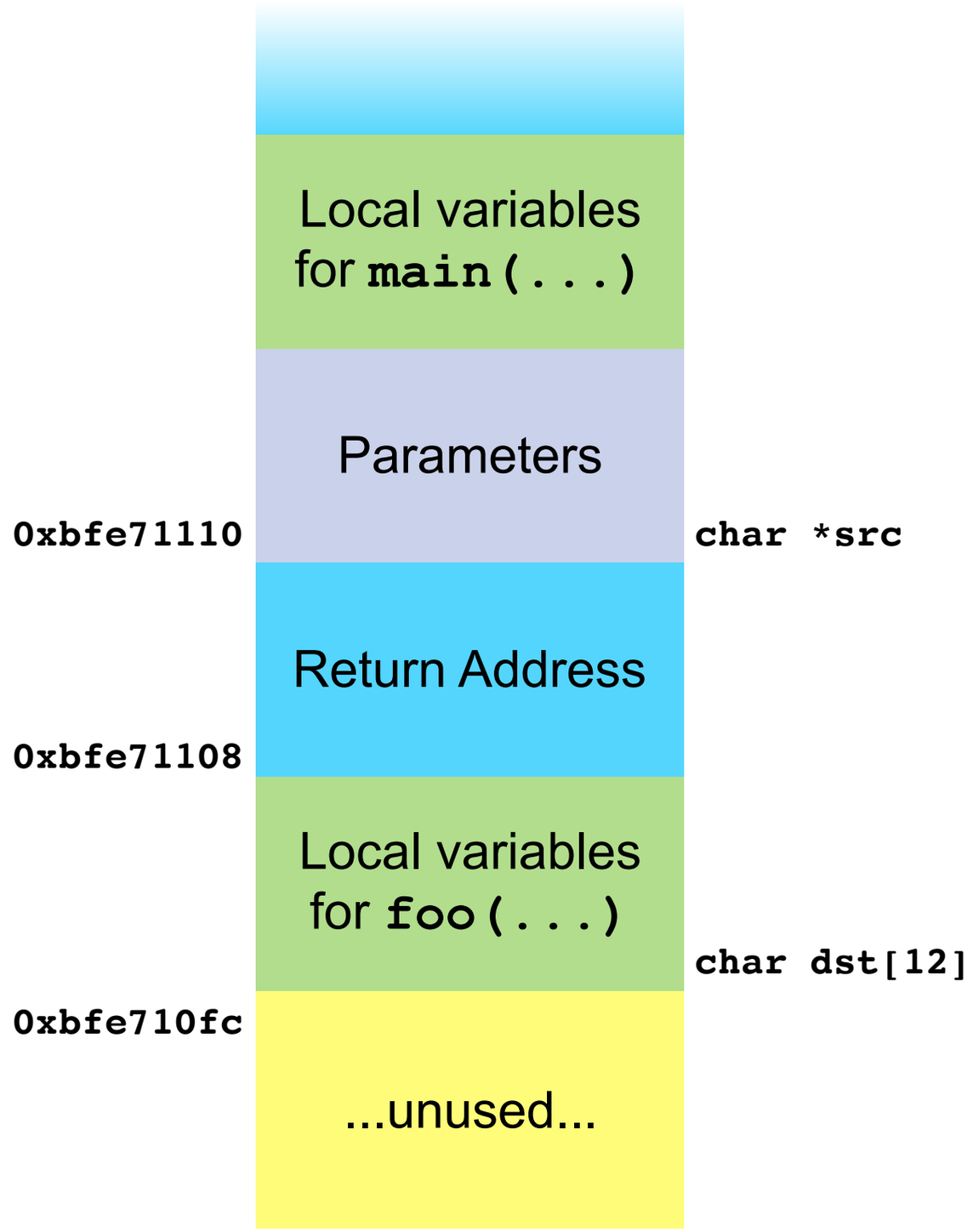
```
// overflow.c
#include <string.h>
#include <stdio.h>

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

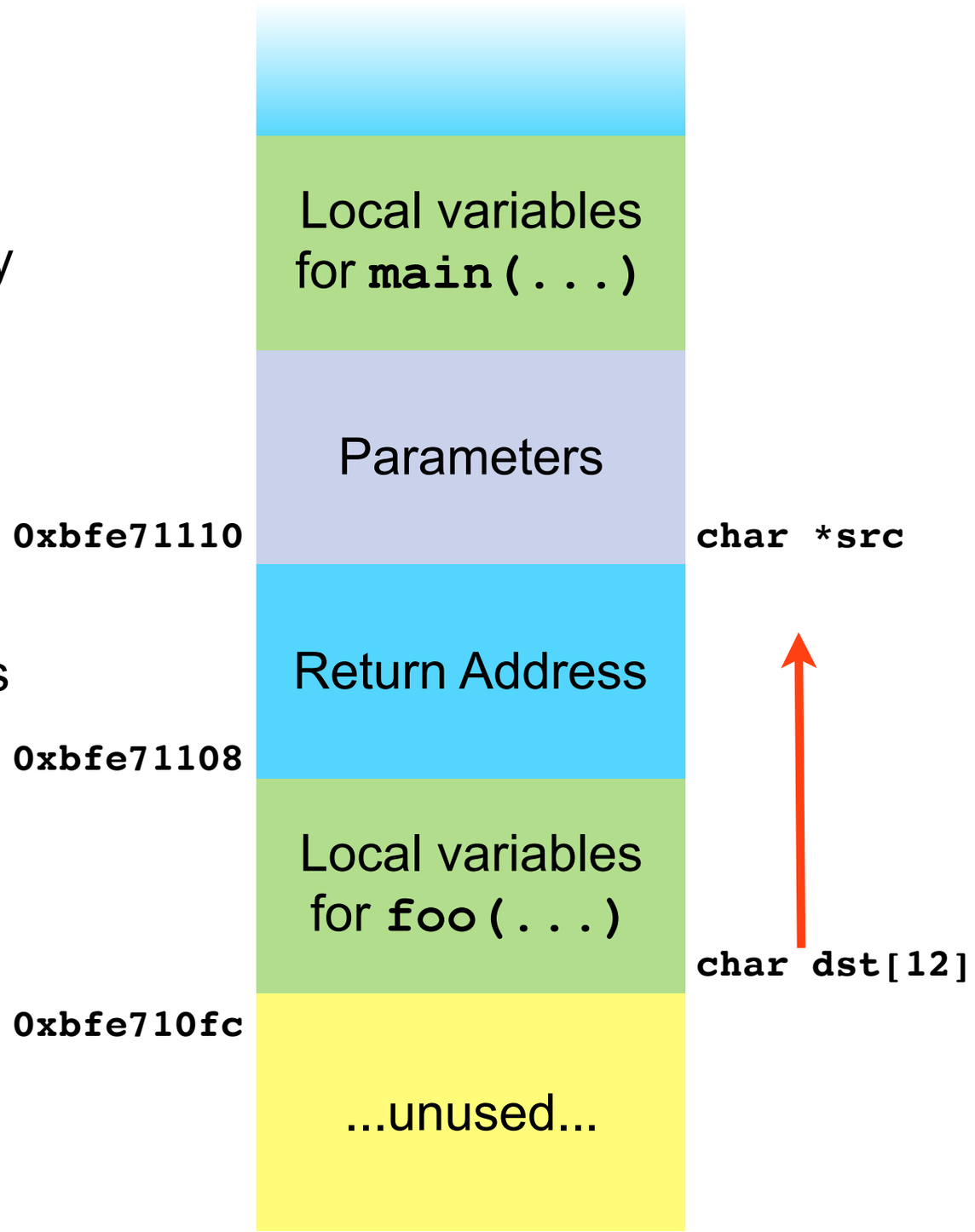
```
// overflow.c
#include <string.h>
#include <stdio.h>

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?