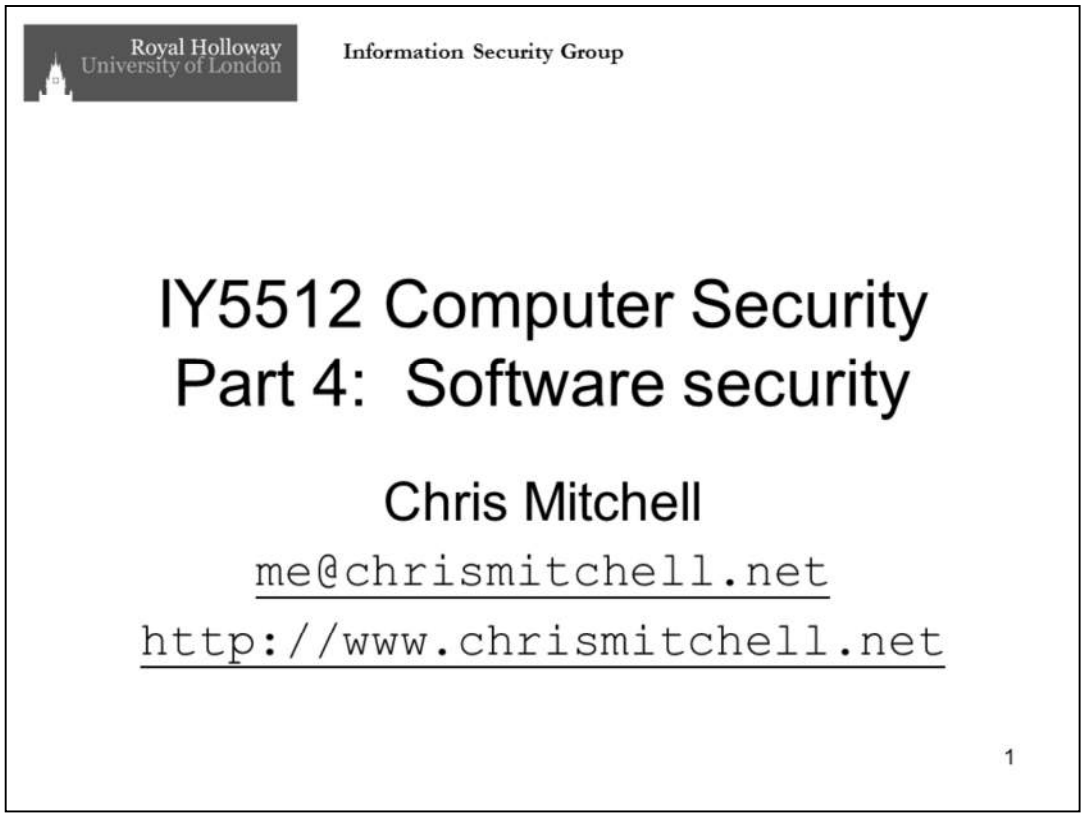
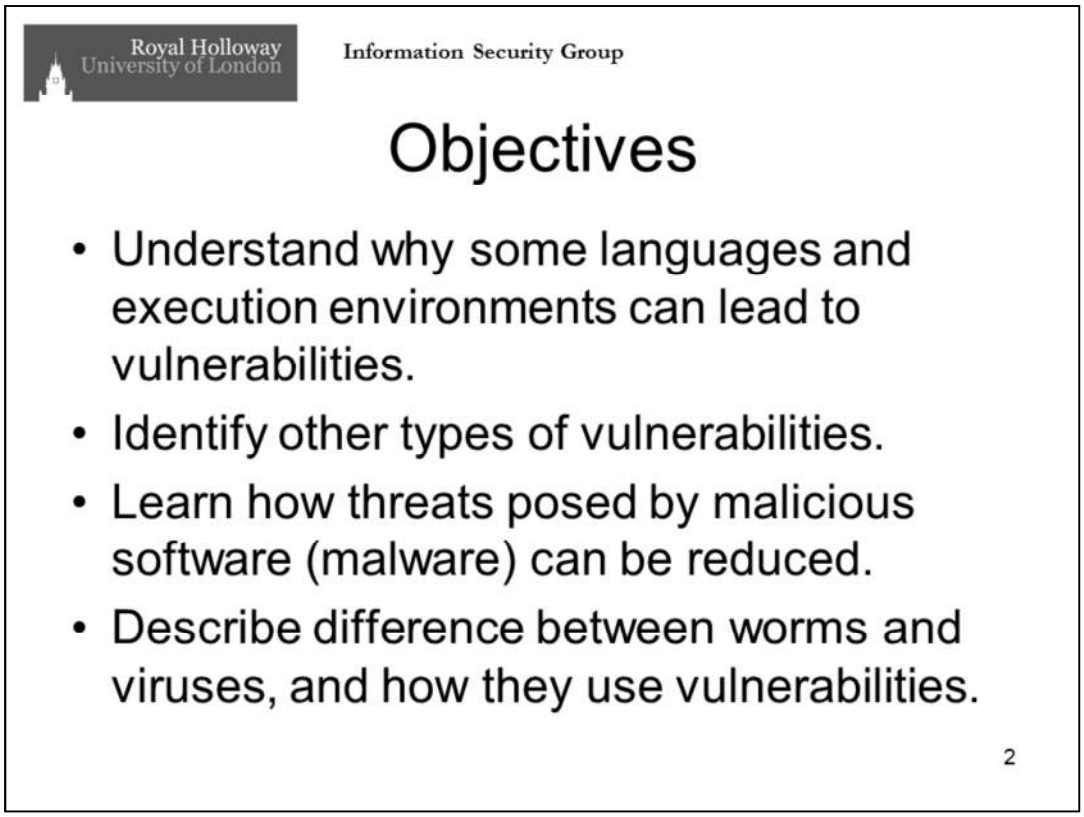

Royal Holloway
University of London
Information Security Group
IY5512 Computer Security
Part 4: Software security
Chris Mitchell
me@chrismitchell.net
http://www.chrismitchell.net
1



The objectives of this part of the course are as follows:

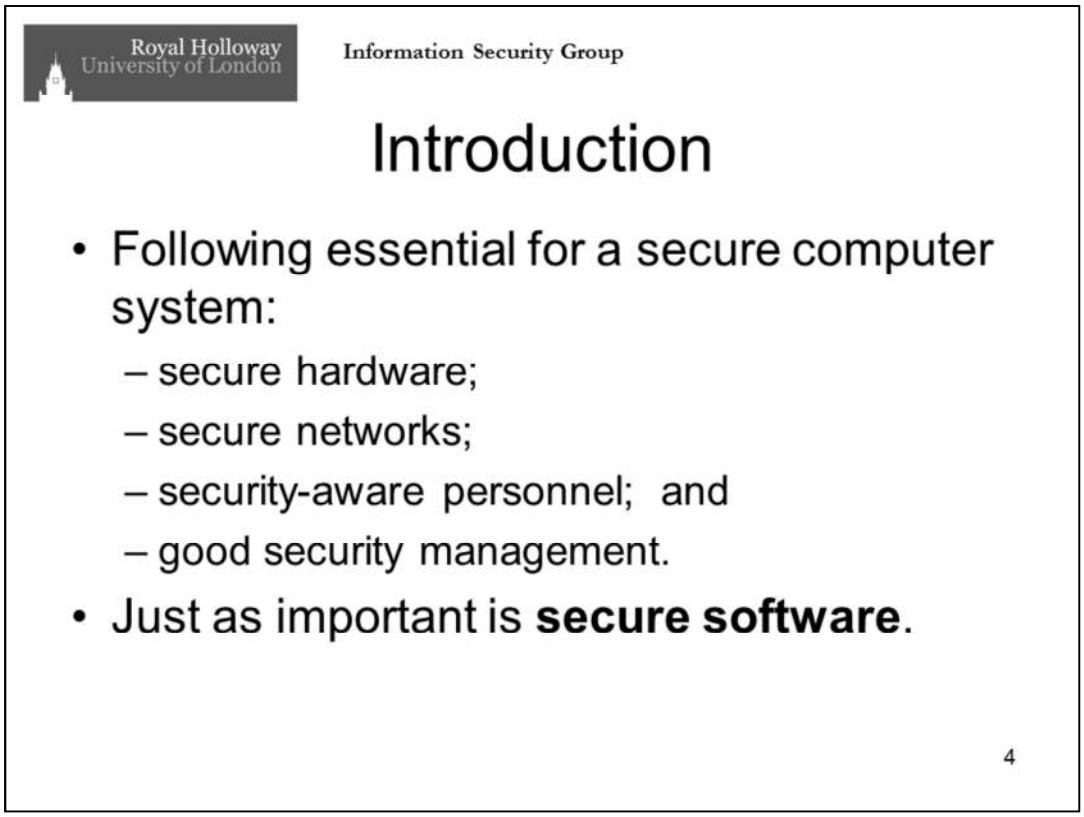
• to understand why certain programming languages and execution environments may lead to vulnerabilities;

• to identify other types of vulnerabilities;

• to learn how the threats posed by malicious software (malware), such as viruses and worms, can be reduced;

• to appreciate the difference between worms and viruses, and how they exploit different types of vulnerabilities.

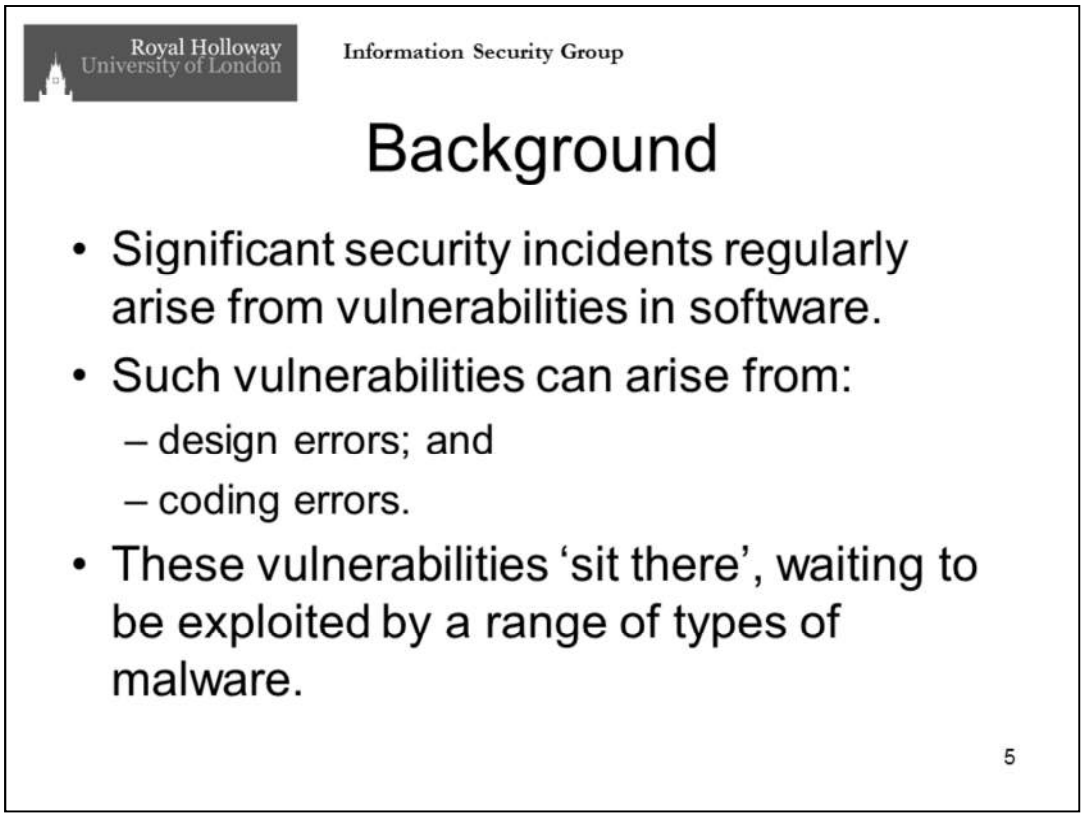
We start this part of the course by reviewing how software can pose a threat to security.



As discussed throughout this course, the following are all essential if a computer system is to be secure:

- secure hardware;
- secure networks;
- security-aware personnel; and
- good security management.

However, also extremely important is *secure software*.

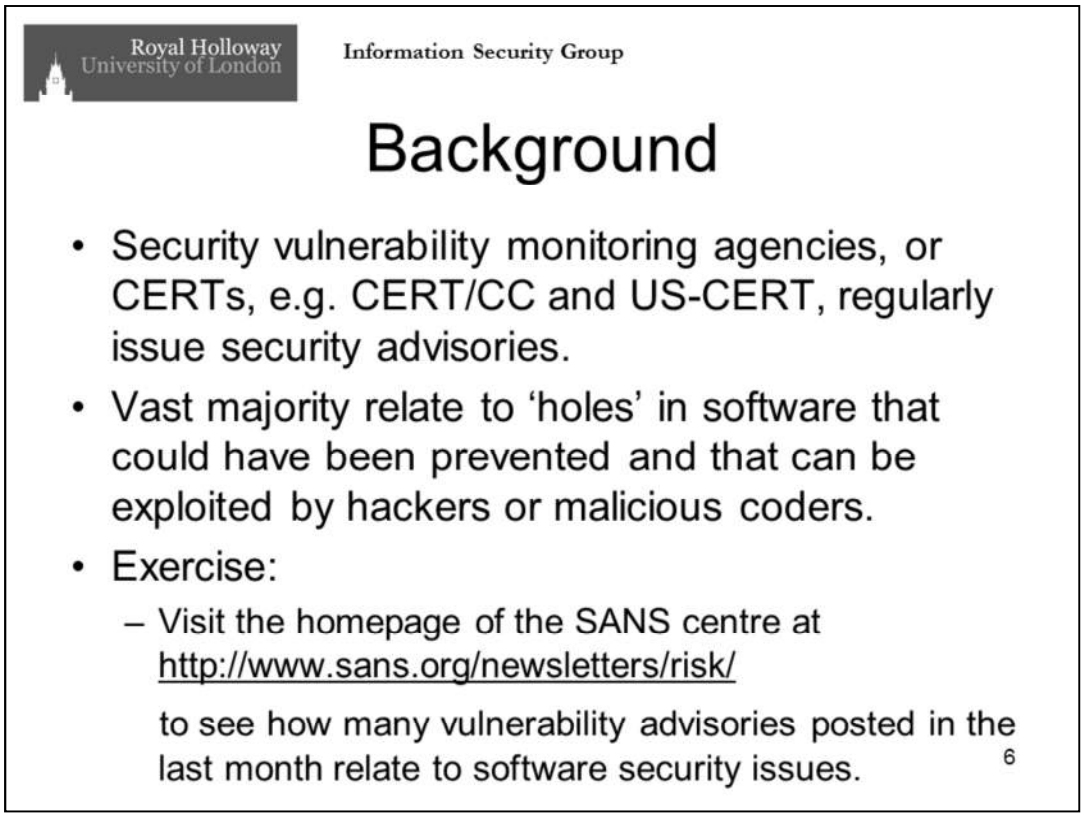


Computer systems regularly encounter significant security incidents arising from vulnerabilities in software. Such vulnerabilities can arise from both:

- design errors; and
- coding (implementation) errors.

These vulnerabilities 'sit there', waiting to be exploited by a range of categories of malware, including worms and viruses.

This does not imply that the world is full of either malicious or incompetent programmers. However it does imply that if more attention was given to the design of secure code, then many security failures could be avoided.



Security vulnerability monitoring agencies (often called CERTs, for Computer Emergency Readiness Teams), such as the CERT Coordination Center (CERT/CC) and the US Computer Emergency Readiness Team (US-CERT), regularly issue security advisories. CERT/CC is part of CERT, based at Carnegie-Mellon University in the US. The CERT/CC home page is here:

http://www.cert.org/certcc.html

US-CERT is art of the US Department of Homeland Security. The US-CERT home page is here:

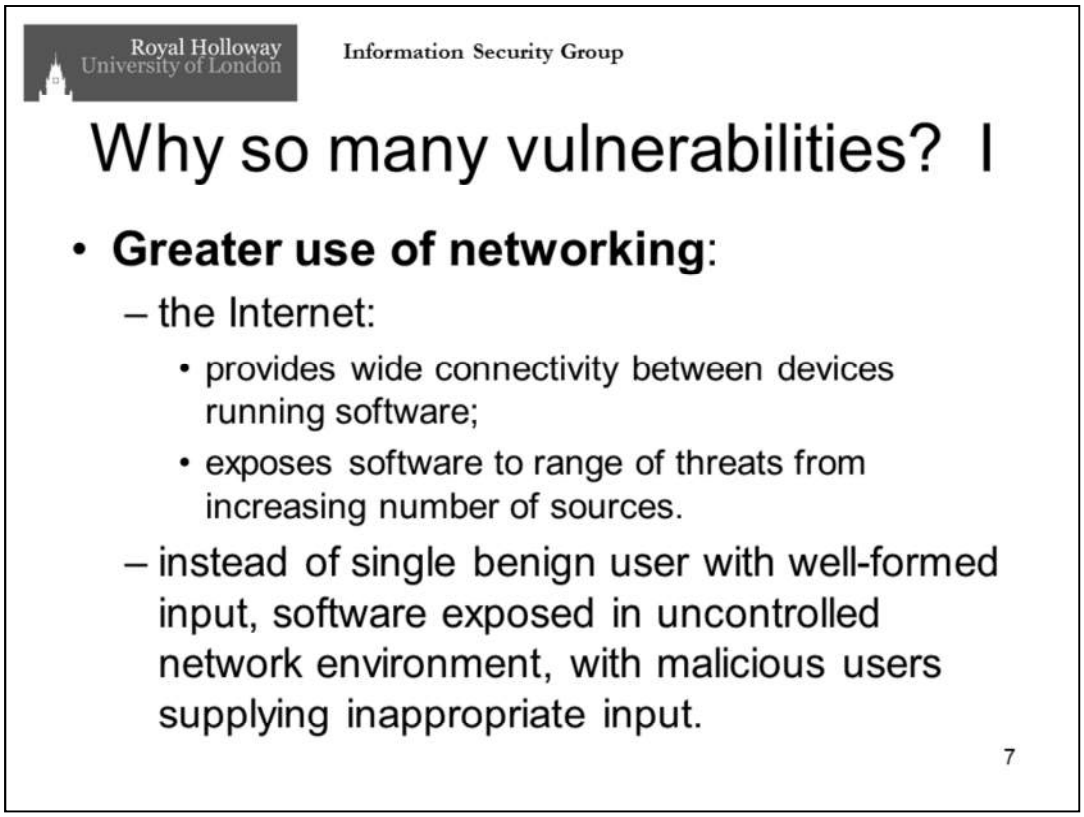
http://www.us-cert.gov/

The vast majority of these advisories relate to 'holes' in software that could have been prevented, and that can be exploited by hackers or malicious coders.

As an exercise visit the homepage of the SANS centre at:

http://www.sans.org/newsletters/risk/

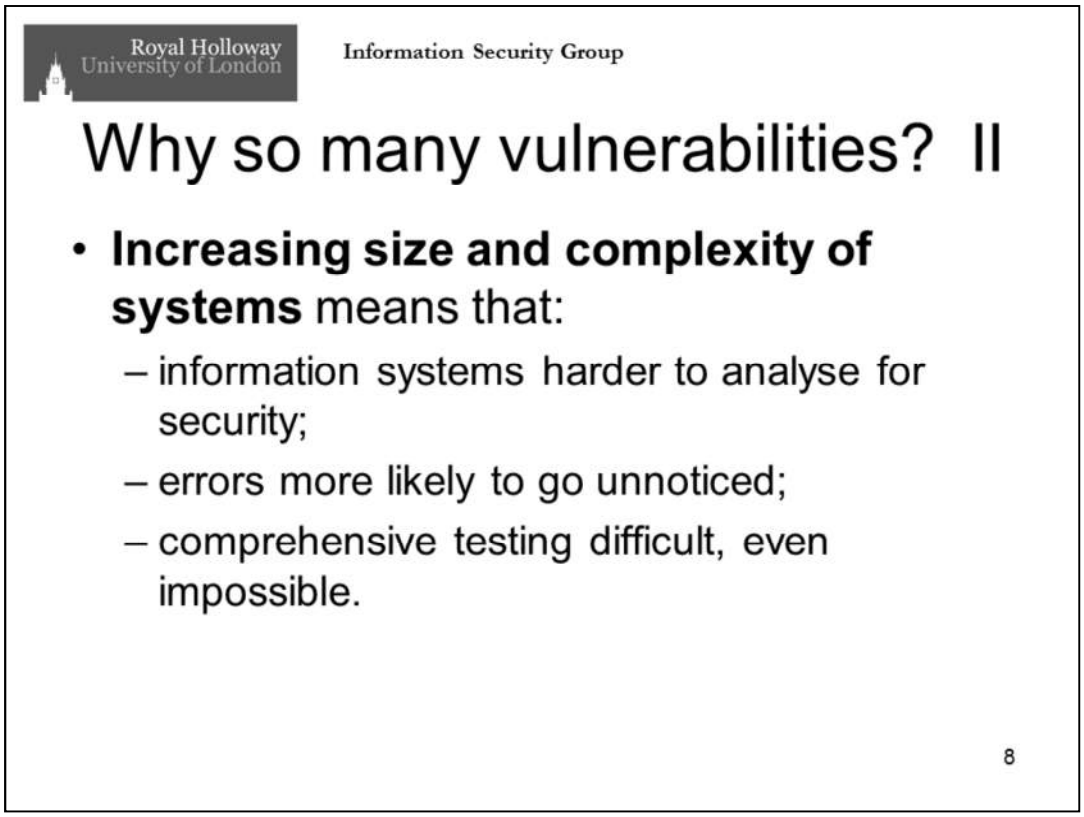
to see how many vulnerability advisories posted in the last month relate to software security issues. [SANS is a commercial organisation].



There are a variety of reasons for the huge number of software vulnerabilities being found, reported and (we hope) fixed. One reason is the ubiquitous use of networking technology (and the Internet in particular). The success of the Internet:

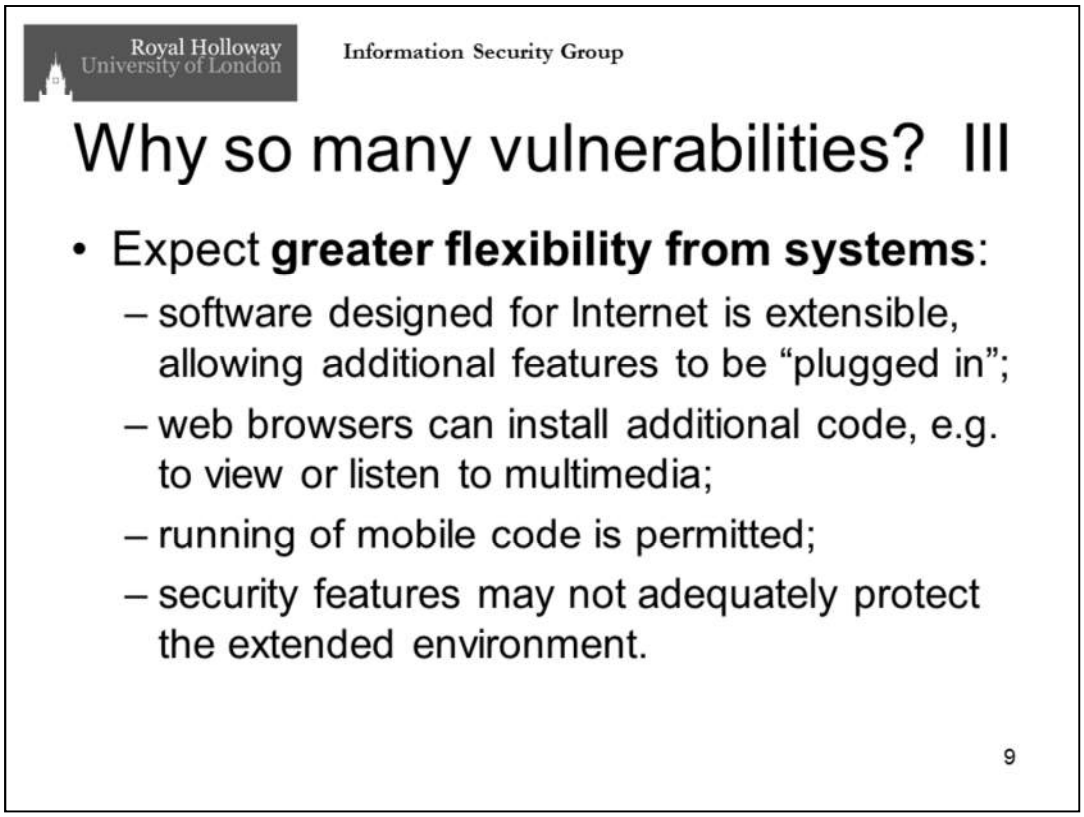
- provides greater connectivity between devices that run software;
- means that software is more widely exposed to a range of threats from an increasing number of sources.

Instead of a single benign user providing the software with well-formed input, software is exposed in an uncontrolled network environment, with malicious users capable of supplying inappropriate input.



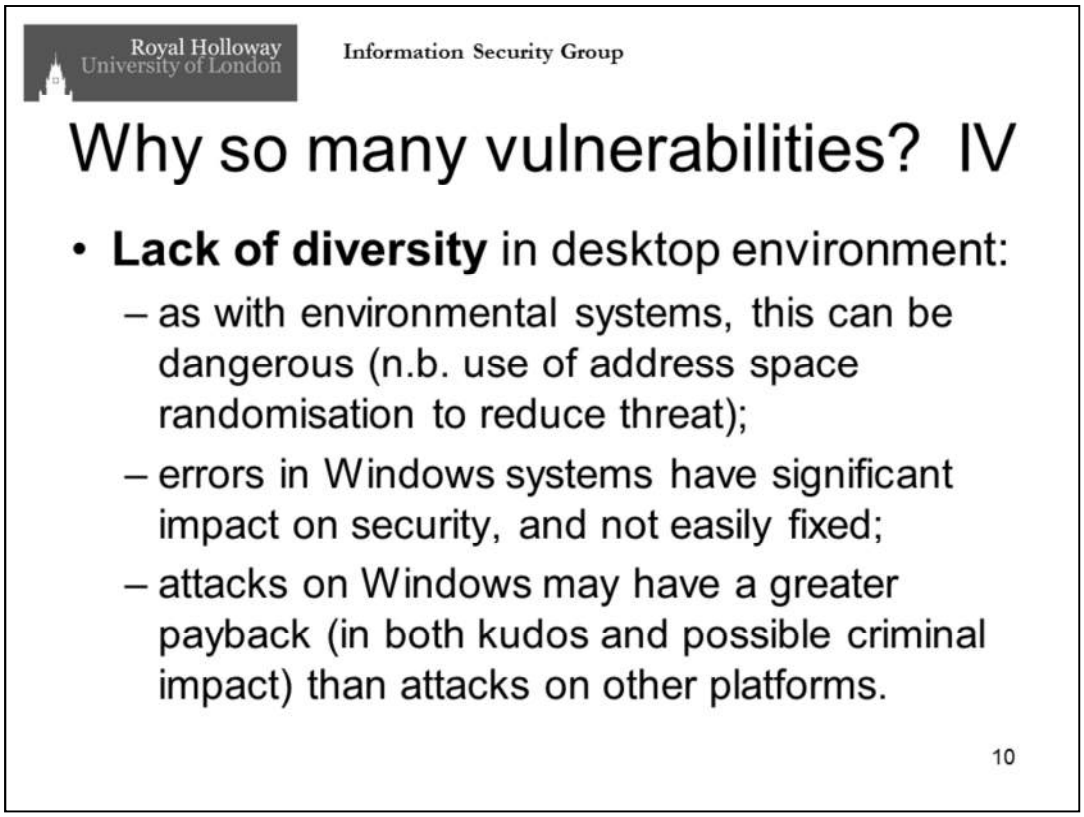
A second reason for vulnerabilities is the ever-increasing size and complexity of systems. This complexity means that:

- information systems are harder to analyse for security;
- errors are more likely to go unnoticed;
- comprehensive testing is difficult and, in practice, often impossible to conduct.



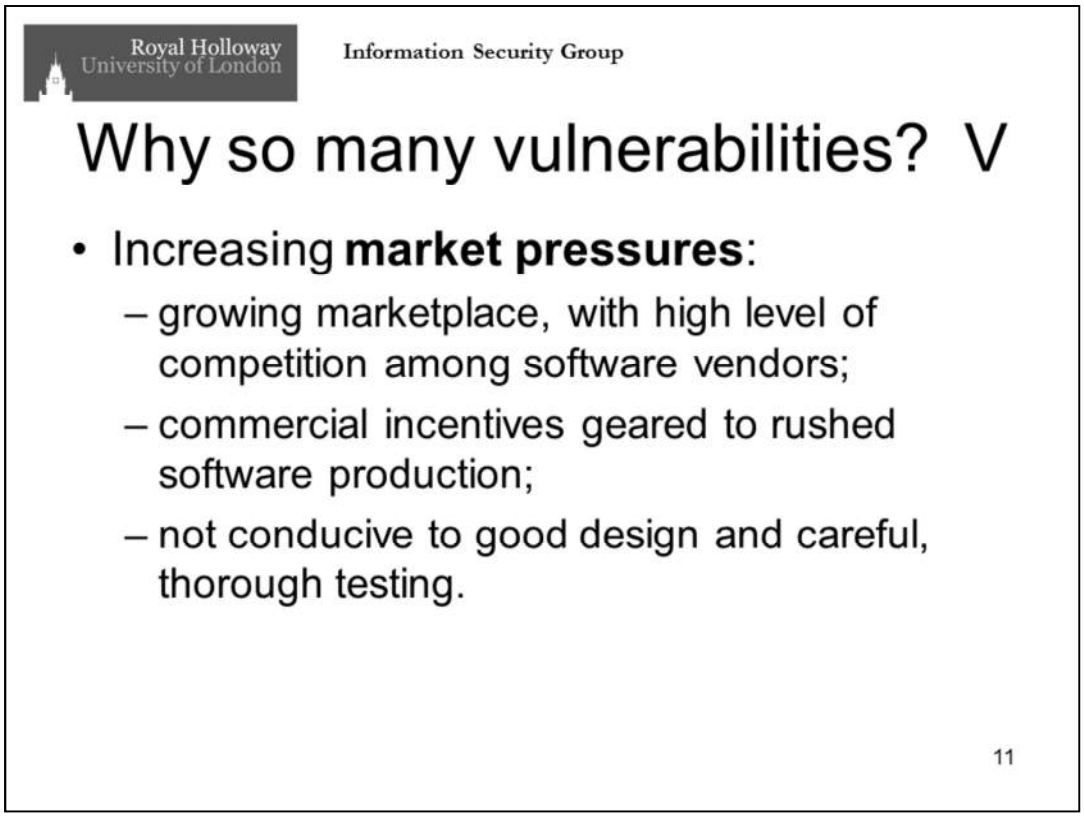
A third reason is that we expect greater flexibility from systems:

- software designed for the Internet is extensible, allowing for additional features to be "plugged in";
- web browsers can install additional code, e.g. for viewing or listening to multimedia;
- the execution of mobile code (i.e. code downloaded by to an end PC by a server) is permitted;
- the provided security features, perhaps designed before the need for additional flexibility was envisaged, may not adequately protect the extended environment.



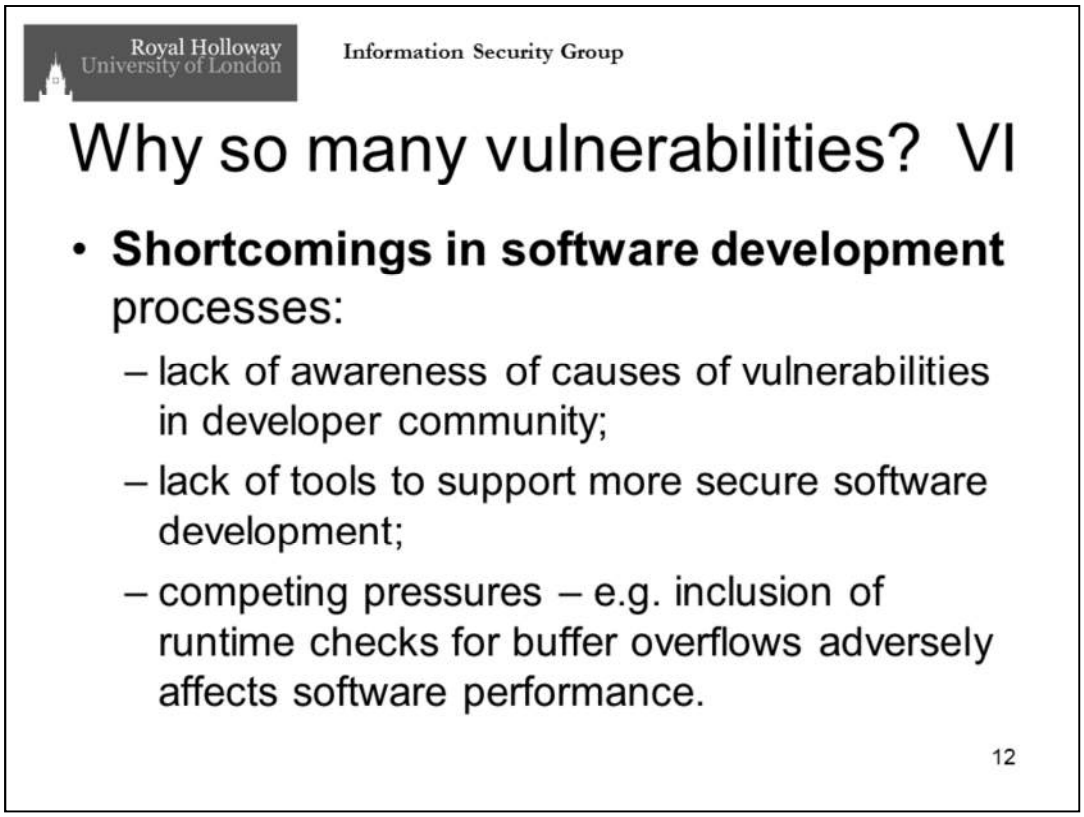
A fourth reason is the lack of diversity in the computing environment – Windows is dominant on the desktop, and the only other significant OS is Unix (including MacOS):

- as with environmental systems, this lack of diversity can be dangerous (Windows 8 includes measures designed to increase the diversity of certain internal system properties, i.e. the address space, of a running system);
- errors in Windows systems have a significant impact on security and are not easily fixed;
- attacks on Windows may be seen as having a greater payback (in both kudos and possible criminal impact) than attacks on other platforms.



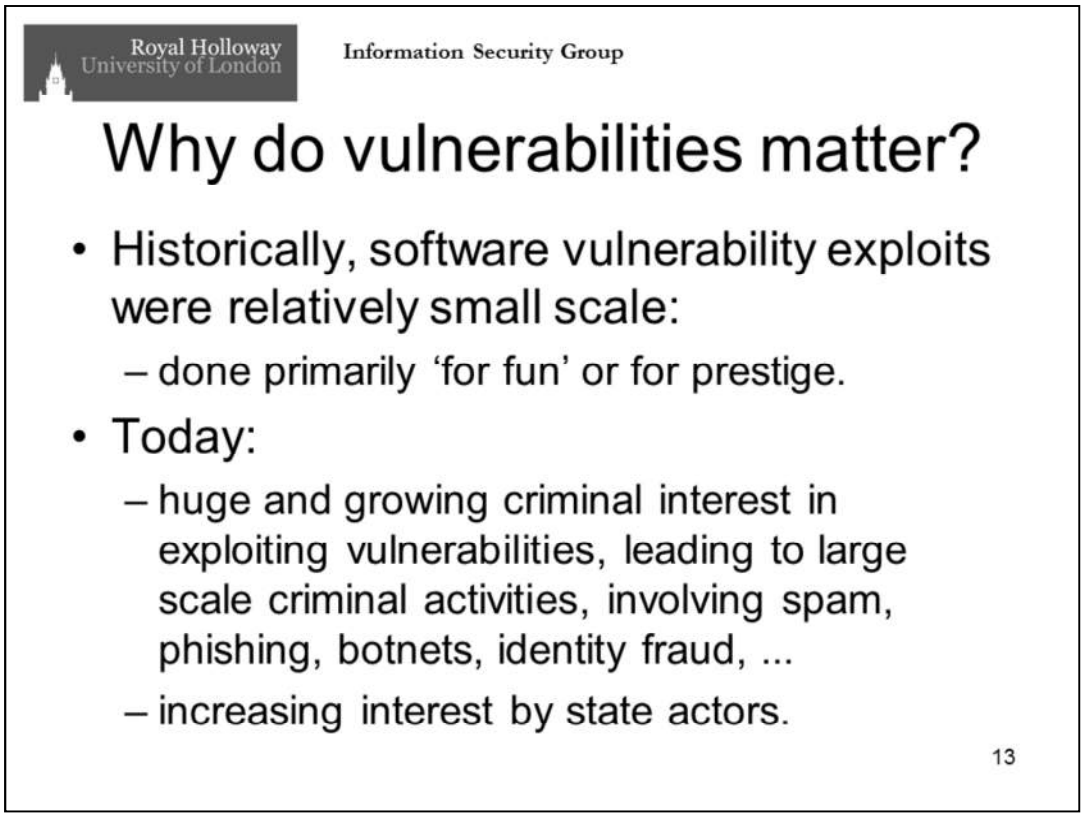
A fifth reason for the presence of vulnerabilities is the ever-increasing market pressures on vendors:

- it is a growing marketplace, with a high level of competition among software vendors;
- commercial incentives are geared towards rushed software production;
- the pressure to ship as early as possible is not conducive to good design and careful, thorough testing.



A sixth reason for the existence of so many vulnerabilities are widespread shortcomings in the software development processes:

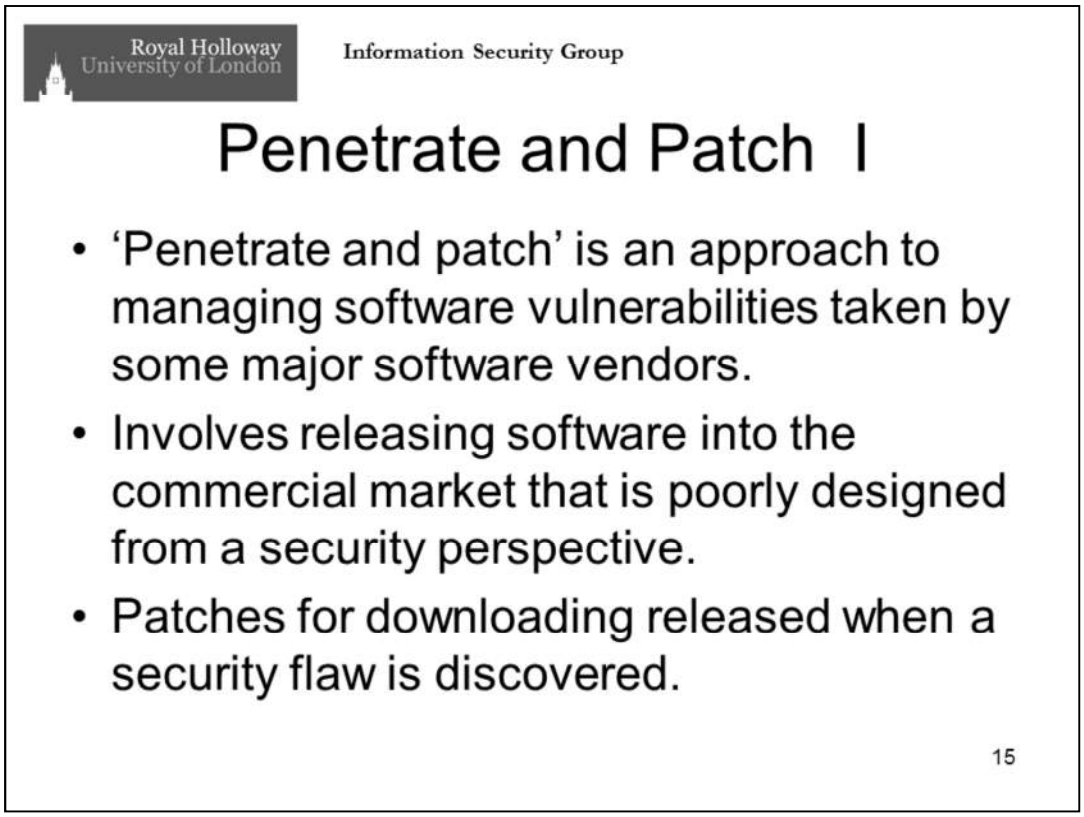
- there is a lack of awareness of the causes of vulnerabilities amongst the wider software developer community (hopefully this situation is changing; for example, security is now becoming a 'standard part' of a Computer Science degree);
- historically there has been a lack of tools to support more secure software development (although this is being addressed, including by free tools such as those available on the SDL site – see: http://www.microsoft.com/security/sdl/adopt/tools.aspx);
- there are competing pressures on software vendors relating to software security – e.g. the inclusion of runtime checks for security issues such as buffer overflows has an adverse impact on software performance.



Historically, exploits of software vulnerabilities were relatively small scale, and were done primarily 'for fun' or for prestige reasons.

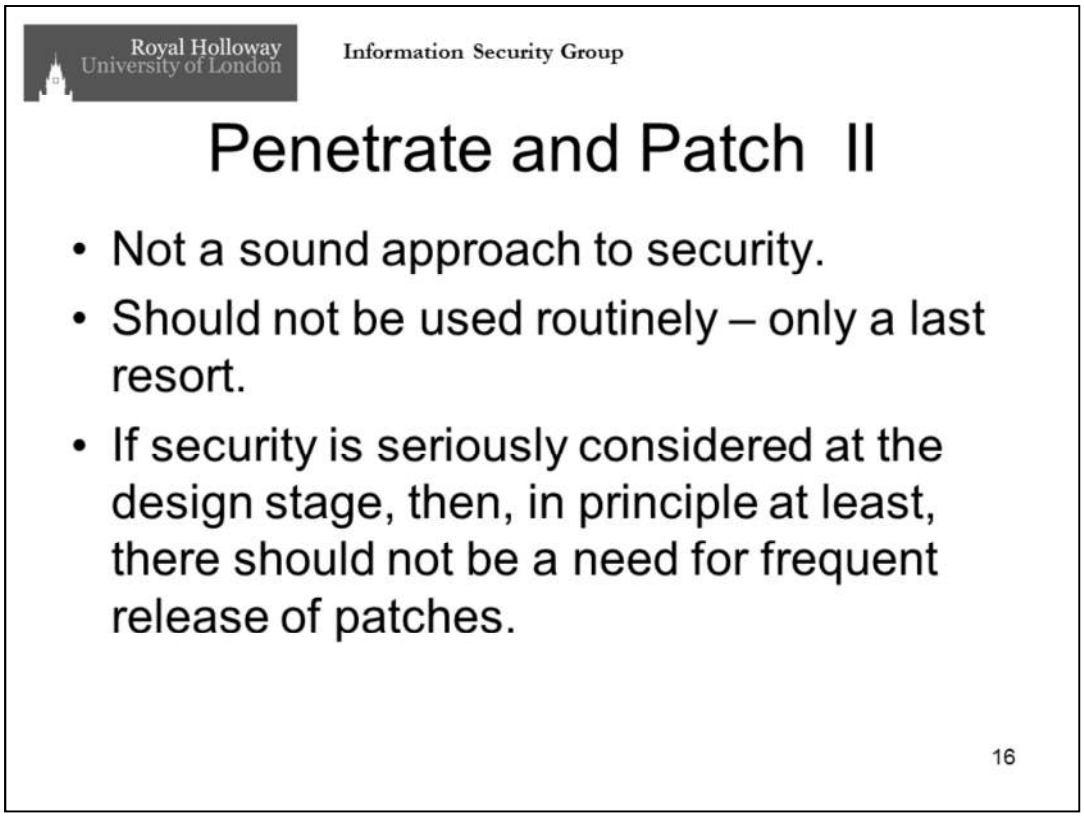
Today, there is a huge and growing criminal interest in exploiting vulnerabilities, giving rise to large scale criminal activities, e.g. involving spam, phishing, botnets, identity fraud, etc. At the same time, nation states are increasingly viewing malware and other types of attack on information infrastructures as ways of conducting espionage and even covert (or overt) war.

We next consider ways in which the existence of software vulnerabilities can be addressed during development.



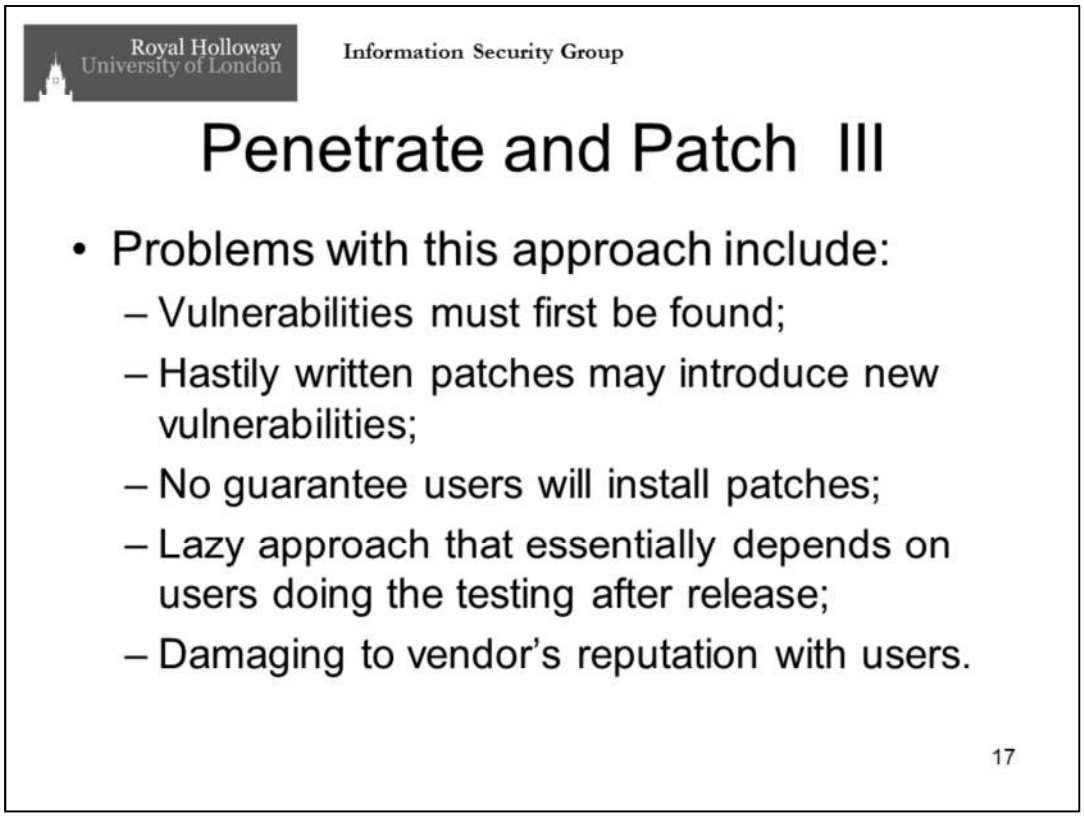
'Penetrate and patch' is an approach to managing software vulnerabilities taken by some (maybe all?) major software vendors. This is typically not a deliberate decision, but one often made in ignorance of the seriousness and scope of vulnerabilities, and the ease with which they can be accidentally introduced into software.

It involves releasing software into the commercial market that is poorly designed from a security perspective. Patches for downloading by users are released when a security flaw is discovered.



Penetrate and patch is clearly not a sound approach to managing software security. It should not be used routinely, but only as a last resort.

If security is seriously considered at the design stage, then, in principle at least, there should not be a need for the frequent release of patches. However, practical experience suggests that removing **all** vulnerabilities from complex software, even when great care is taken, is very difficult to achieve.



There are a number of significant problems with this approach:

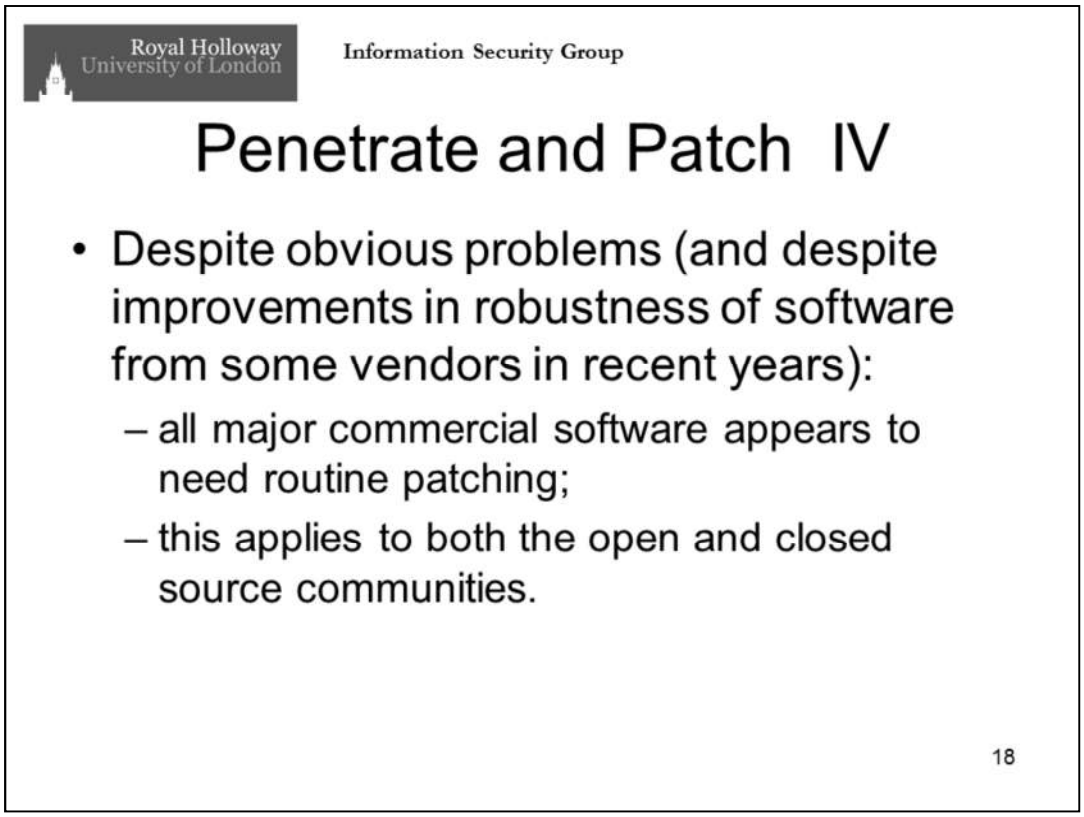
• vulnerabilities must first be found (the worst case is when they are found first by malicious entities who exploit them before any patches/fixes are in place – these are the so called **zero day** vulnerabilities – Flame and Stuxnet exploited zero day vulnerabilities);

• hastily written patches to address newly discovered vulnerabilities may themselves introduce new vulnerabilities;

• there is no guarantee that users will install patches in a timely way;

• this is a lazy approach that essentially depends on users doing the testing after release;

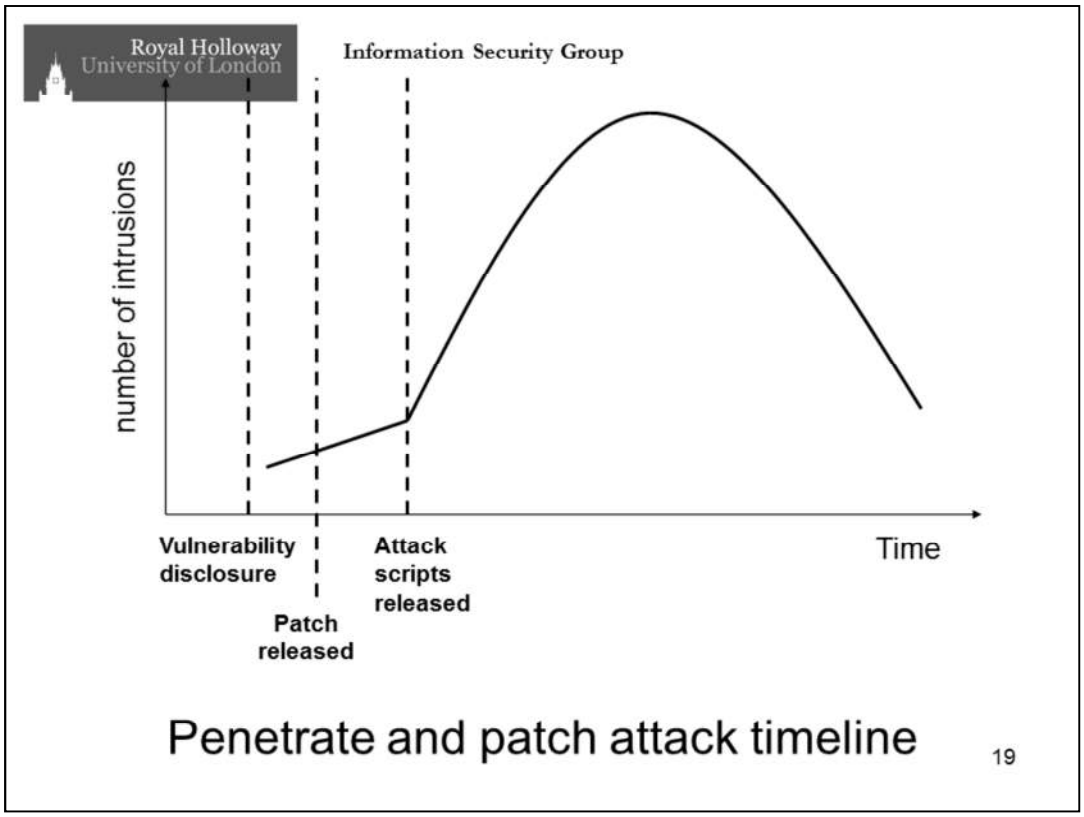
• the need to ship large number of fixes is damaging to a vendor's reputation with users.



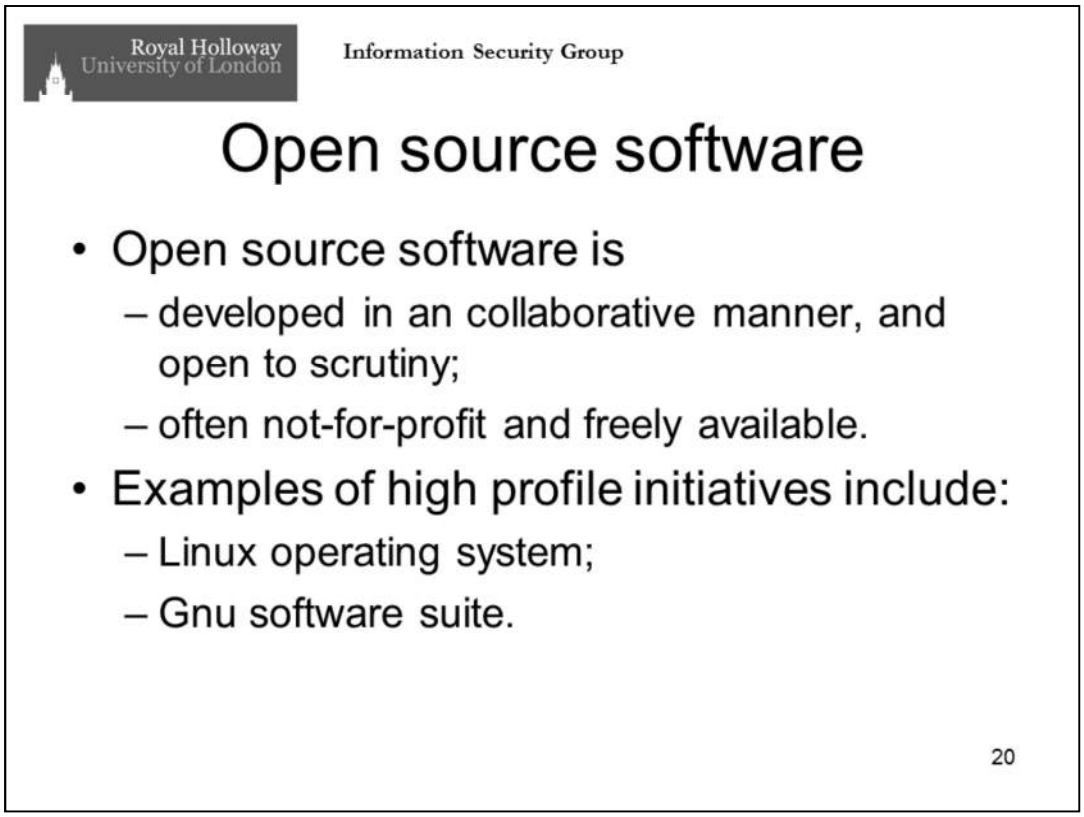
However, despite the obvious shortcomings of this approach (and despite improvements in the robustness of software from some vendors in recent years):

- all major commercial software appears to need routine patching;
- this applies to both the open and closed source communities.

This is despite the fact that organisations such as Microsoft have spent very large sums trying to improve their development processes (and retrain their development staff) to reduce the occurrence of vulnerabilities.

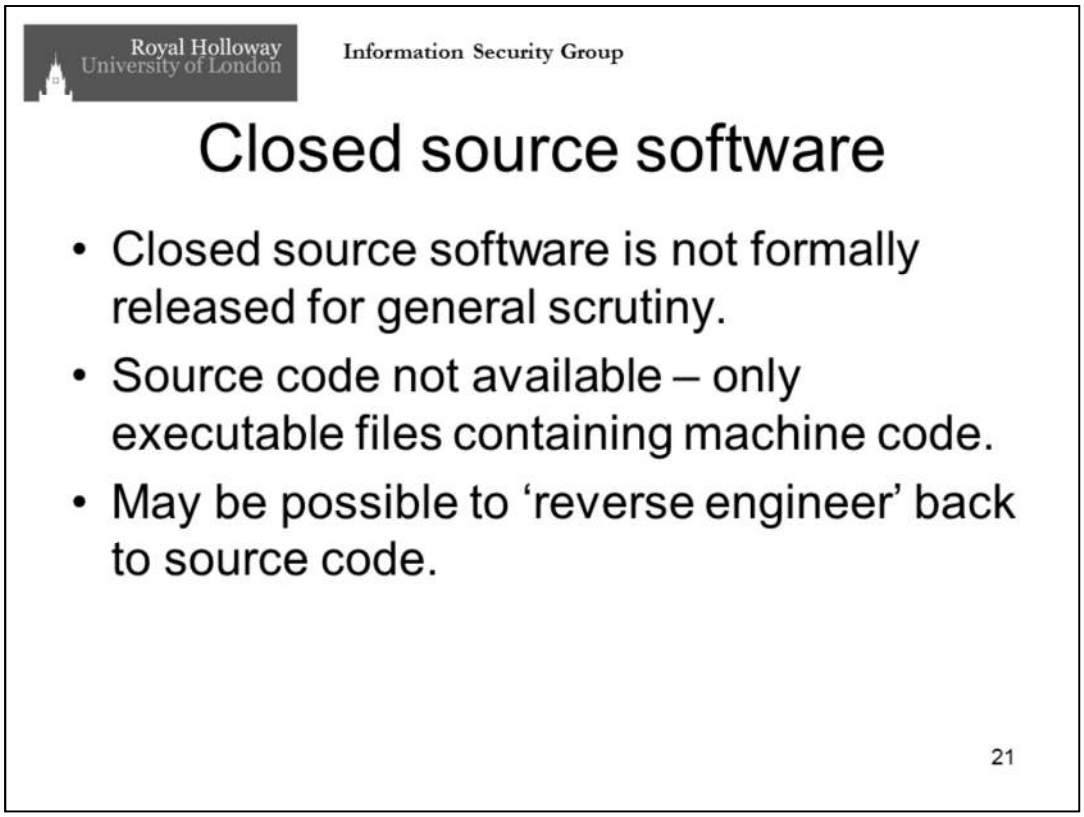


The much quoted figures behind this graph come from a paper published in *IEEE Computer* in 2000. The graph shown is abstracted from data in this paper, and shows a typical intrusion pattern over time. The important points in the timeline are disclosure of a vulnerability, release of a patch and release of an attack script that allows attackers with little technical background to exploit the vulnerability. The results clearly show the problems with penetrate and patch.



Open source software is developed in an collaborative manner, and the software is open to scrutiny. It is often developed on a not-for-profit basis and is freely available.

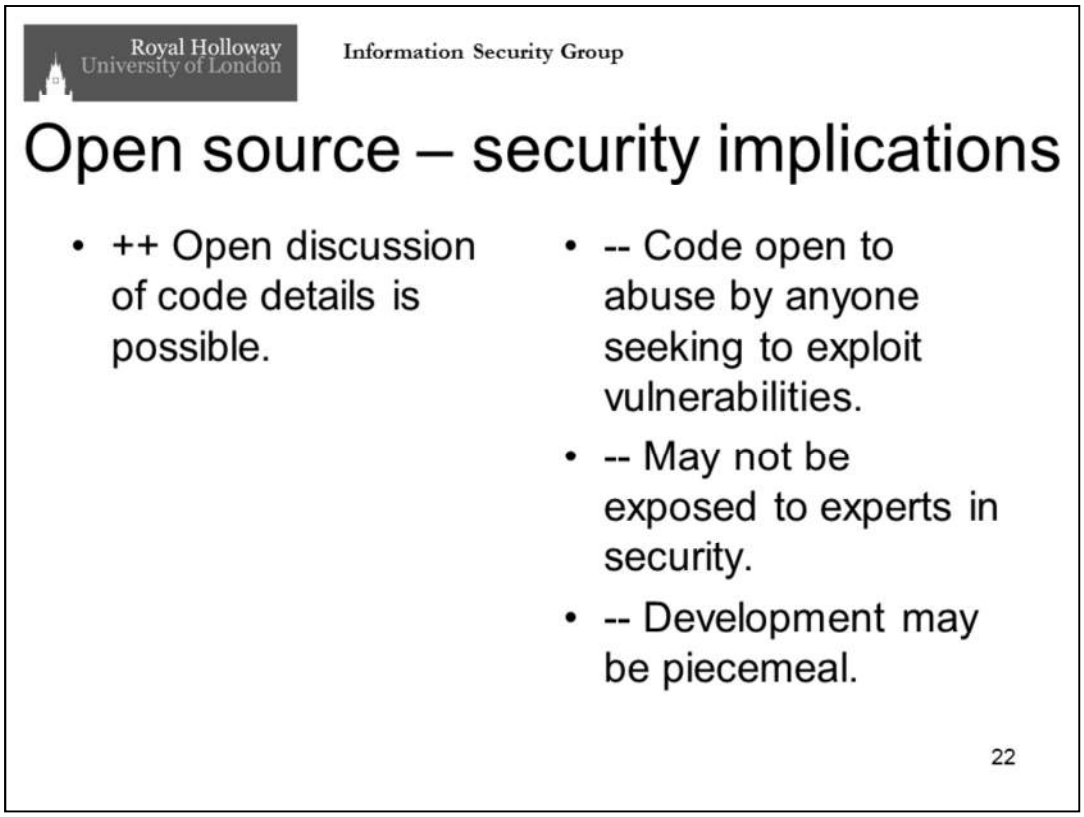
Examples of high profile initiatives include the Linux operating system and the Gnu software suite.



Closed source software is not formally released for general scrutiny. The source code is not publicly available – all that is released are executable files containing machine code.

As with any software, it may be possible to 'reverse engineer' the executable back to source code.

Note that the dichotomy between open and closed source is not as clear as is often assumed. For example, in many cases, vendors of closed source systems will allow major customers access to source code.



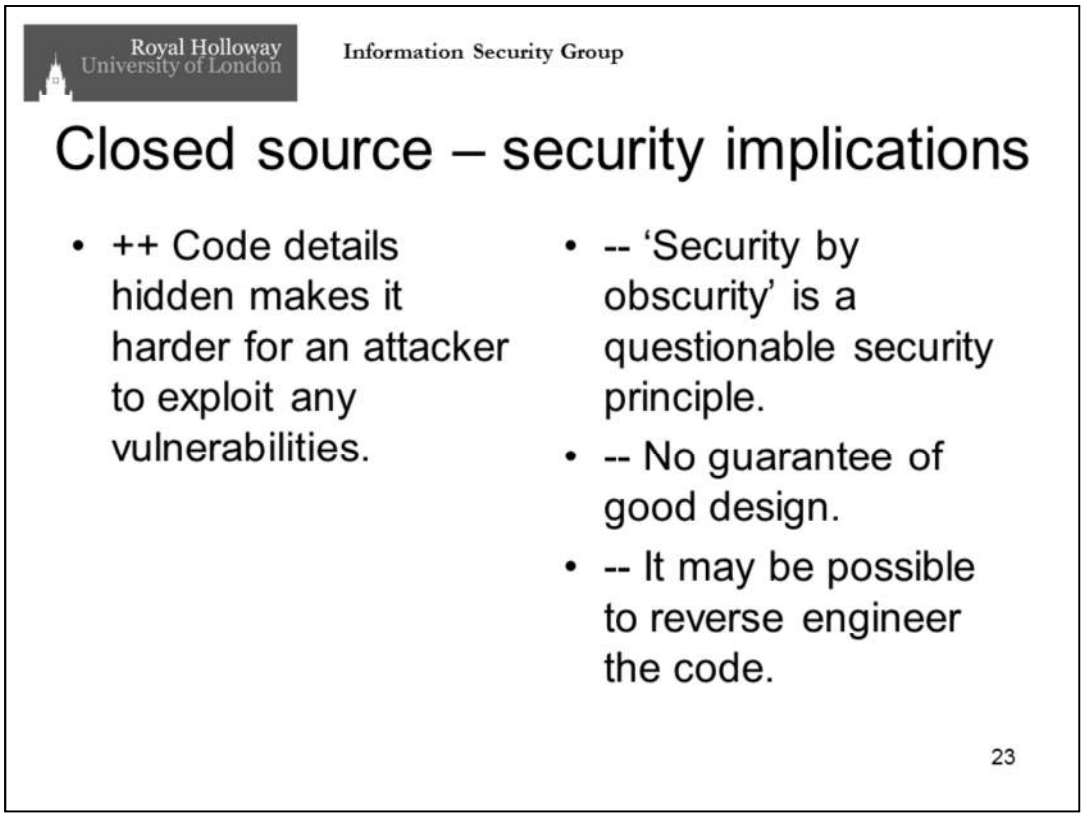
The positive aspects of open source include:

- Open discussion of the code details is possible.

The negative aspects of open source include:

- Code is equally open to abuse by anyone seeking to exploit vulnerabilities;
- The code may not be exposed to experts in security during development;
- Development may be piecemeal.

It could be argued that the last issue is not inherent to open source. It is more a question of whether anyone is committed to maintaining the software. Typically, closed source software is developed and maintained by an organisation which has a vested interested in maintaining it (including fixing bugs when found). Open source software is typically not maintained by a commercial organisation (although there are exceptions), in which case maintenance depends on the willingness and availability of individuals prepared to devote their time to the task. As a result, the level of maintenance can be very patchy – for example, whilst the Linux kernel is very well maintained, other operating system components and applications may often be much less well-maintained.

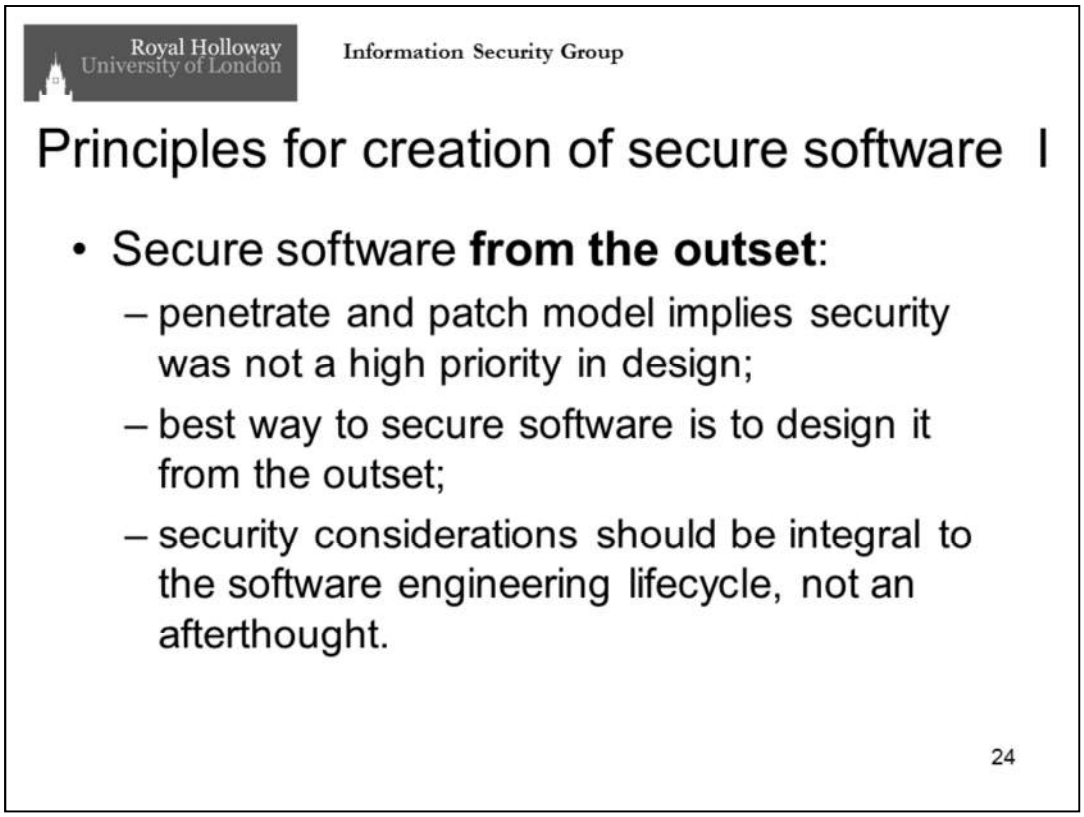


The positive aspects of closed source include:

- The fact that the code details are hidden makes it harder for an attacker to exploit any vulnerabilities.

The negative aspects of closed source include:

- Security by obscurity is a questionable security principle (although almost all security relies on secrets of some kind).
- There is no guarantee of good design.
- It may anyway be possible to reverse engineer the code.



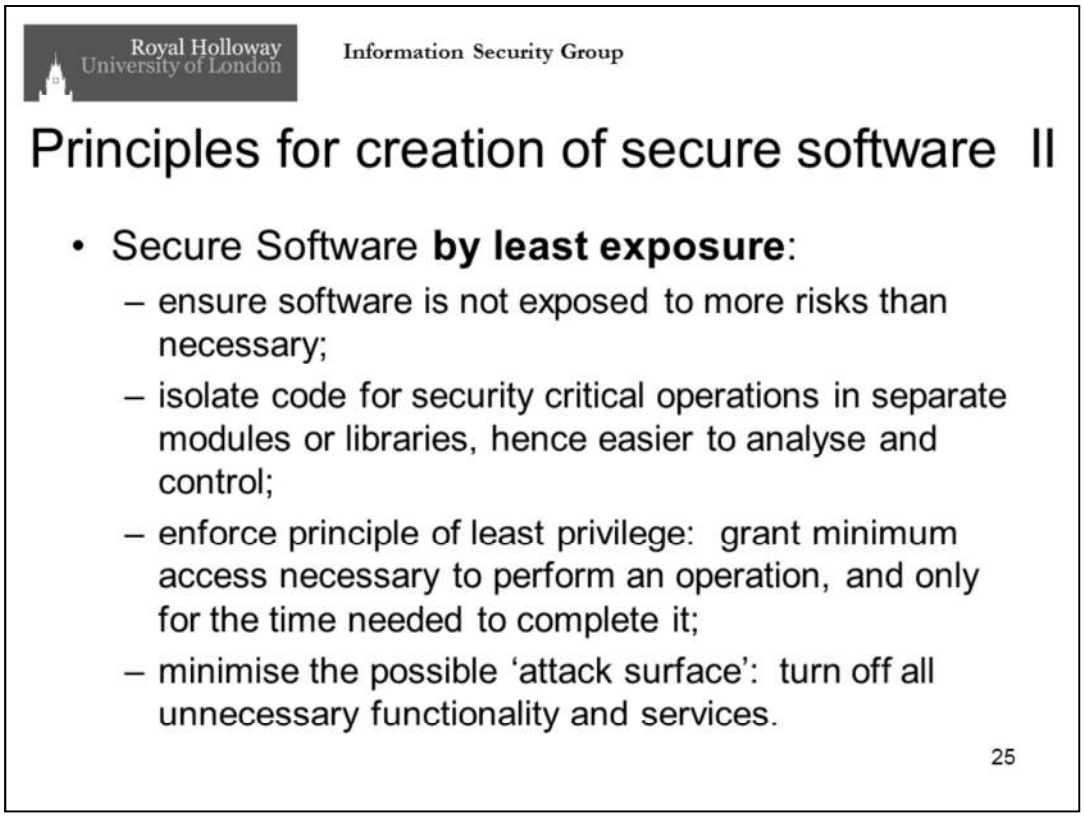
Secure software **from the outset**.

To comply with the software security principle 'secure software from the outset' it is necessary to commence any software development process with a thorough analysis of the security requirements of the intended application.

Adding features to an existing design in order to correct vulnerabilities does not address the underlying problem; either the security requirements were incorrectly specified or the implementation did not meet the security requirements.

Capturing security requirements is essentially a risk assessment exercise.

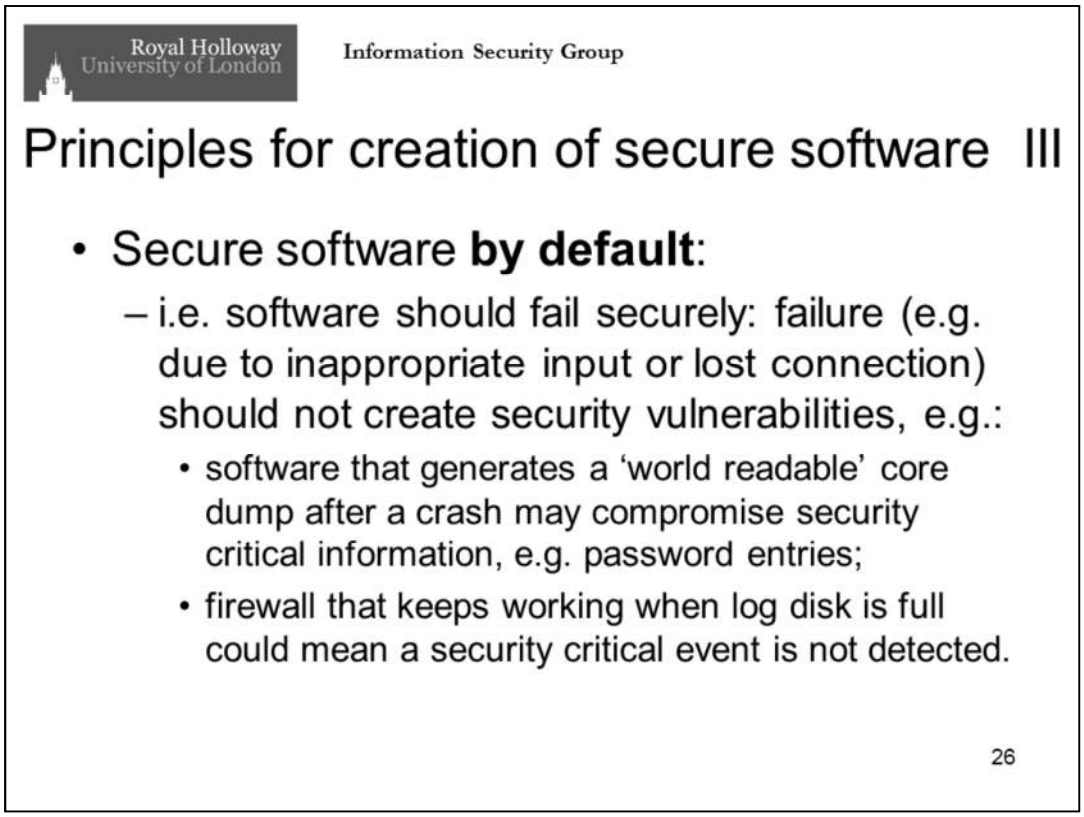
An example of a methodology designed to try to achieve this goal is the Security Development Lifecycle (SDL) model.



Secure Software **by least exposure** means the following:

- ensure that software is not exposed to more risks than necessary;
- isolate code for security critical operations in separate modules or libraries, making it easier to analyse and control;
- enforce the principle of least privilege: grant the minimum access necessary to perform an operation, and only for the time needed to complete it;
- minimise the possible 'attack surface' by turning off all unnecessary functionality and services.

This links closely to the Saltzer-Schroeder security principle of **Least privilege**.

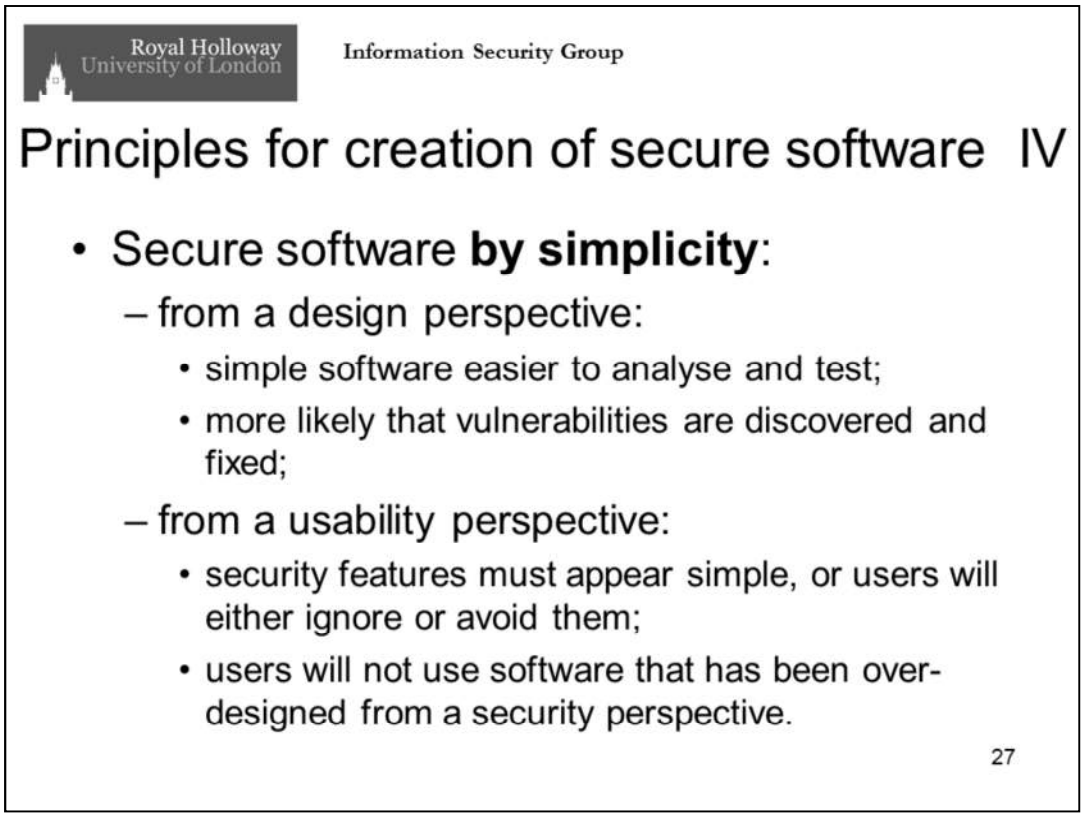


Secure software **by default** means that software should fail securely; that is a failure of the software (e.g. due to inappropriate input or a lost connection) should not expose the system to security vulnerabilities.

For example:

- software that generates a 'world readable' core dump after a crash is problematic, because the core dump may contain security critical information such as password entries;
- a firewall that continues to operate when the disk on which it stores its log becomes full could lead to a failure to detect a security critical event.

This links closely to the Saltzer-Schroeder security principle of **Fail-safe defaults**.



Unfortunately, complex requirements rarely lead to simple software!

Secure software **by simplicity**.

From a design perspective:

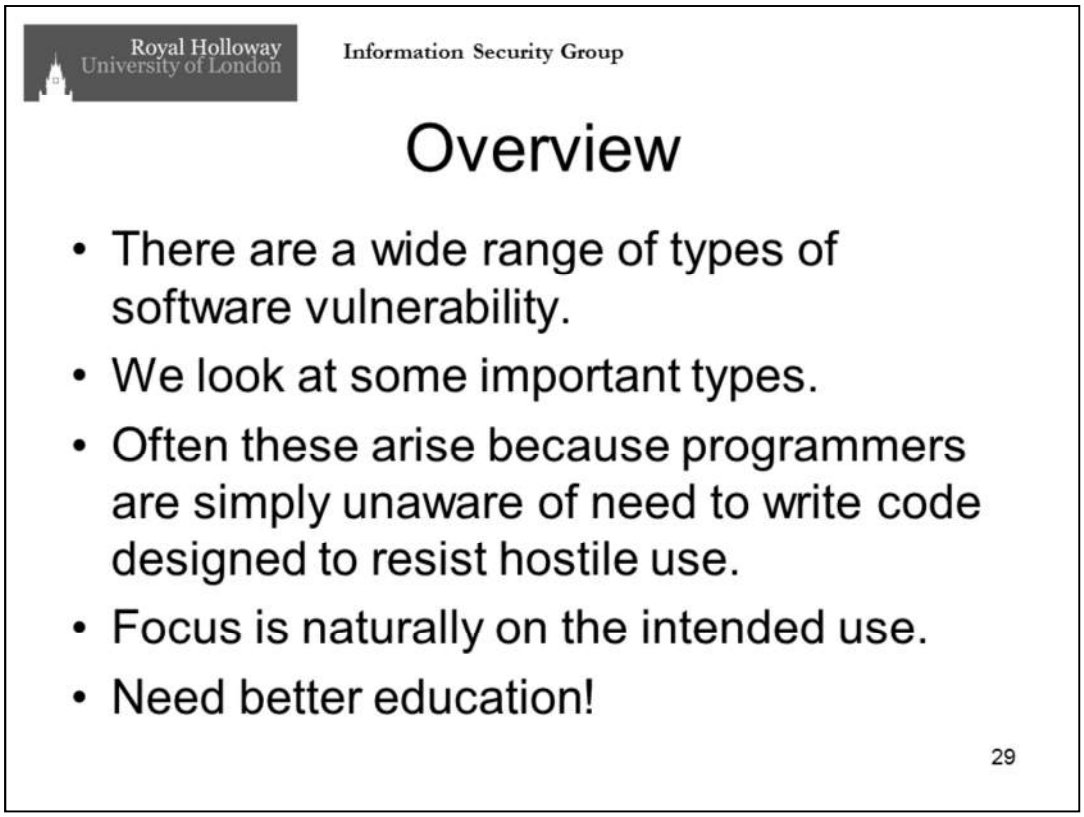
- simple software is easier to analyse and test;
- it is more likely that vulnerabilities will be discovered and corrected.

From a usability perspective:

- security features must appear simple, or users are likely to either ignore or avoid them;
- users will be reluctant to use software that has been over-designed from a security perspective.

This links closely to the Saltzer-Schroeder security principles of **Economy of mechanism** and **Ease of use**.

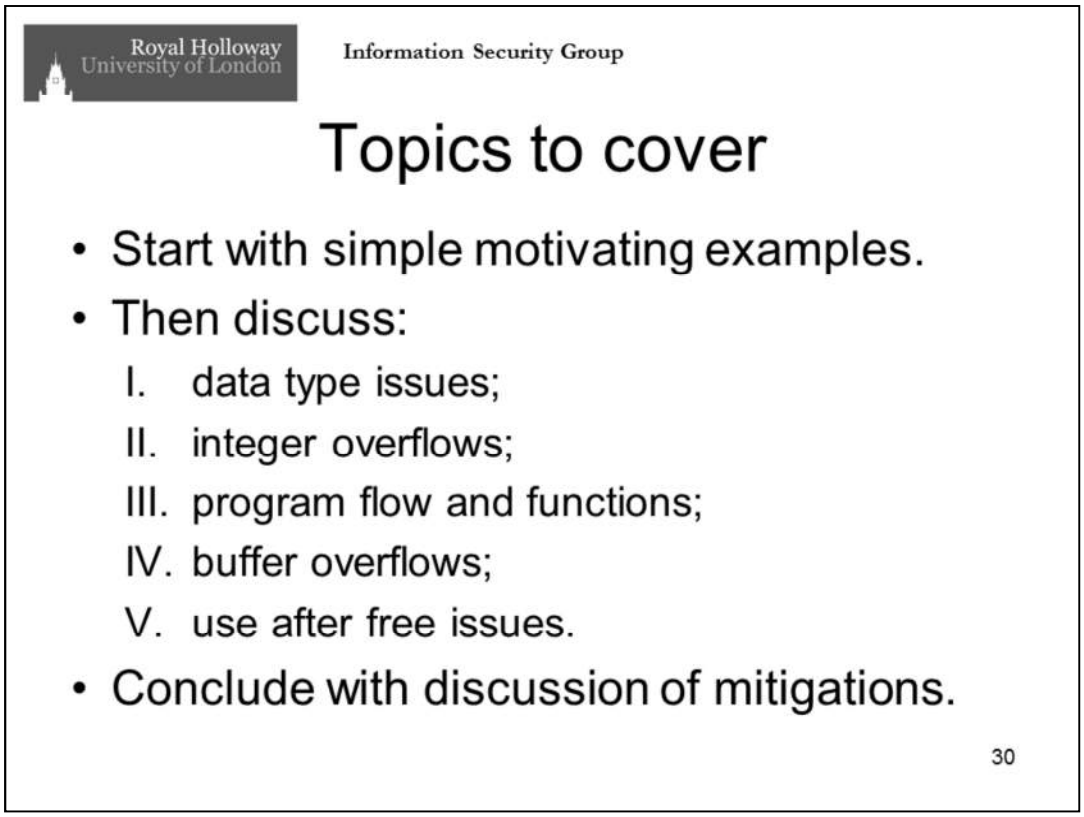
We next consider how vulnerabilities can arise in software.



There are a wide range of types of software vulnerability. We look at some particularly significant categories of such vulnerabilities.

These vulnerabilities often arise because programmers are simply unaware of the need to write code designed to resist hostile or inappropriate use. That is, the focus of the programmer tends to be on the intended use of the software, not on how it might be abused.

This is to a large extent a failure in education and training. It is all too easy when writing code to think about the intended use, and to ignore the possibility of unintended use (e.g. inputs which are not in the expected form).



After a discussion of some simple (motivating) examples, we divide the discussion of vulnerabilities under the following sub-headings:

I. data type issues;

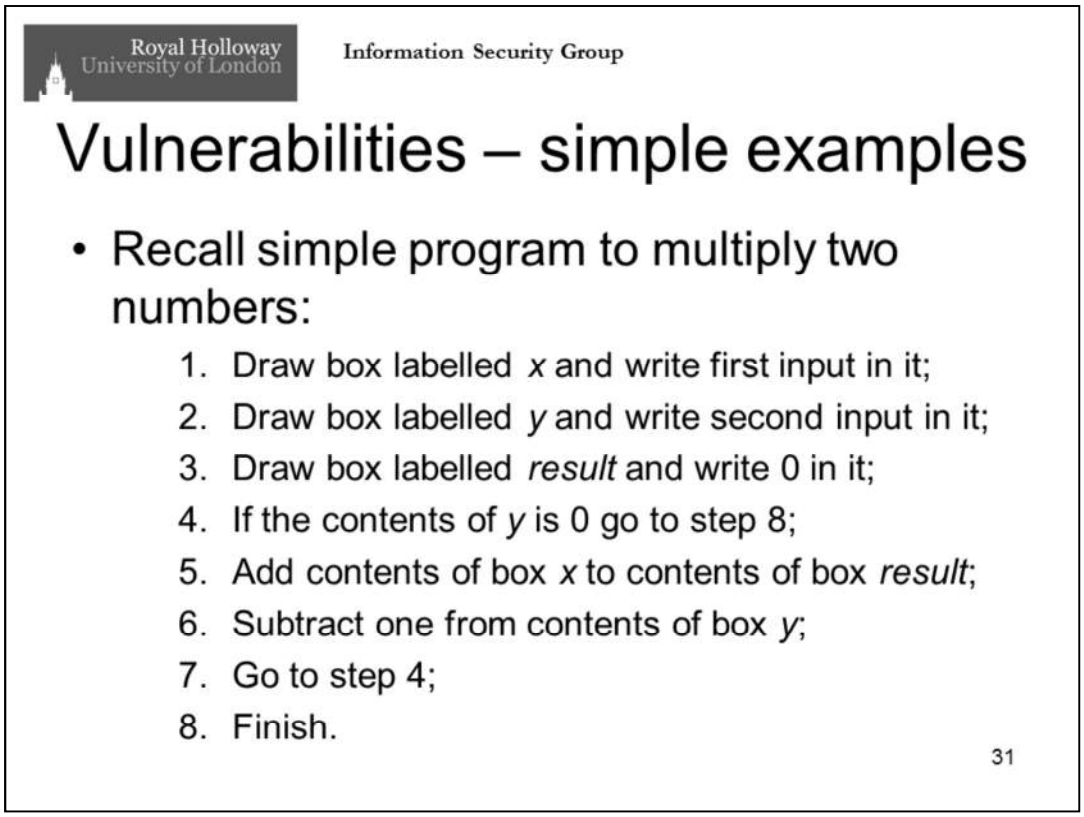
II. integer overflows;

III. program flow and functions;

IV. buffer overflows;

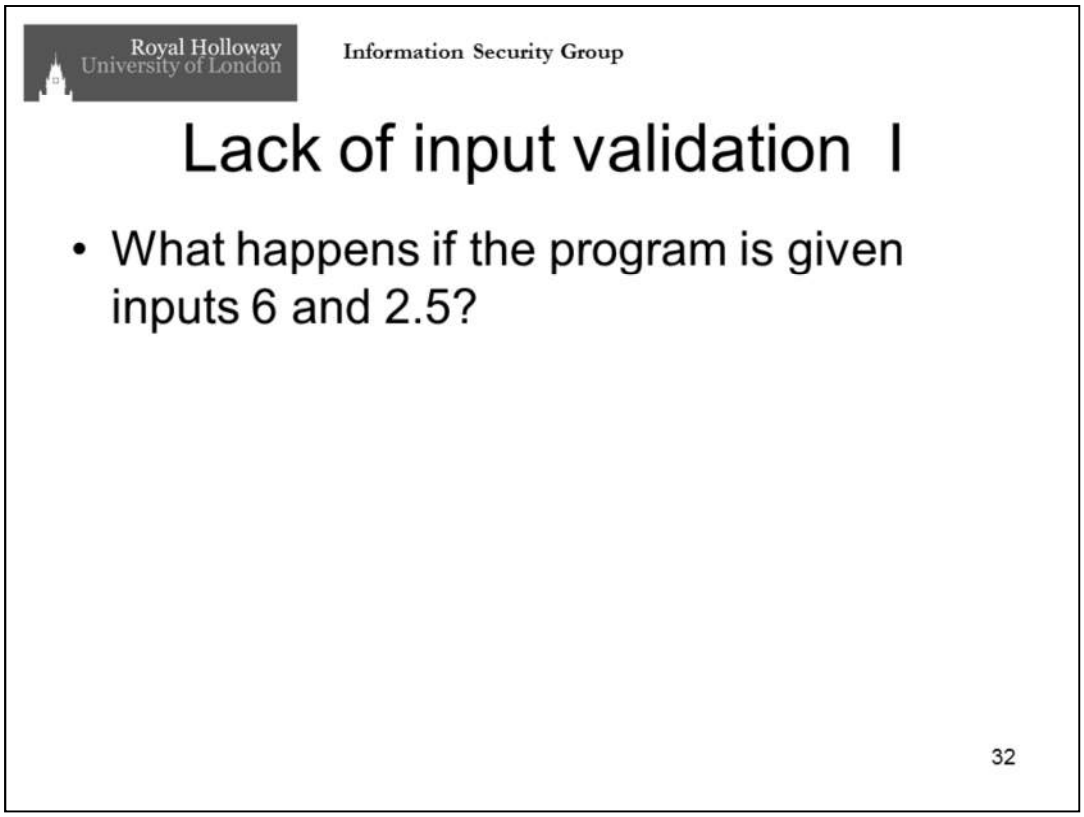
V. use after free issues.

We conclude this discussion with a brief consideration of possible mitigations, i.e. ways of addressing the threat posed by these vulnerabilities.



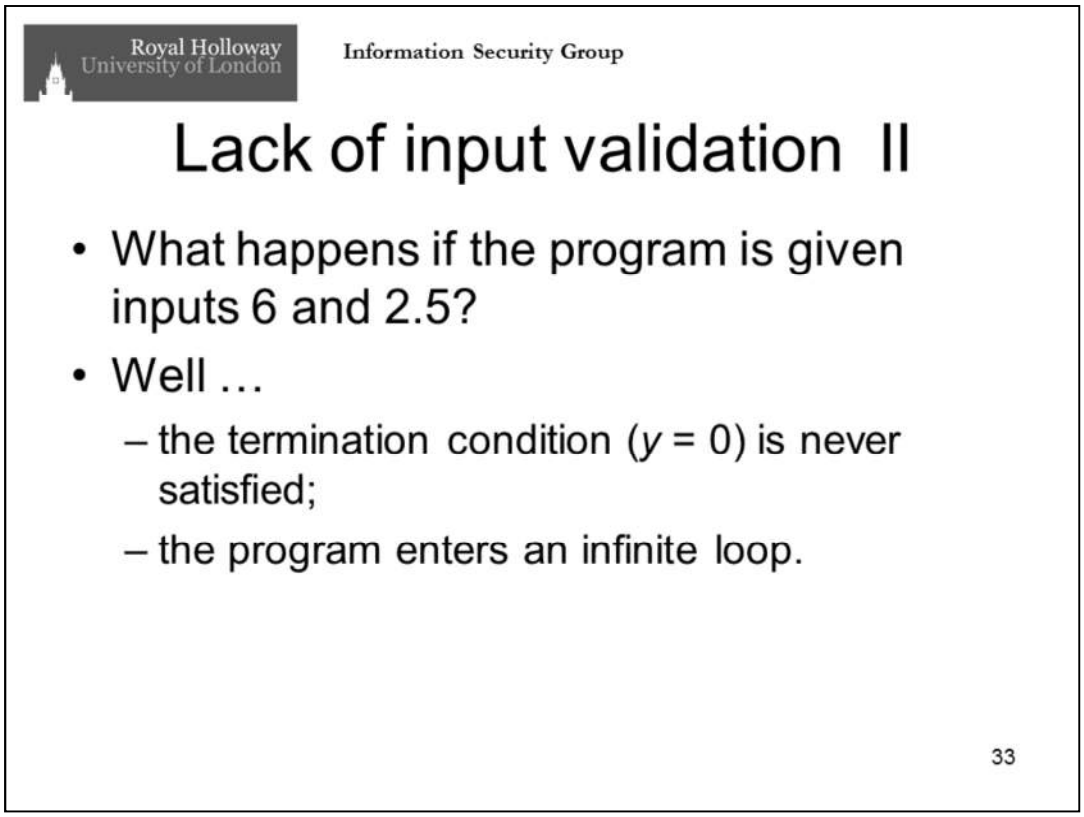
Recall the simple program to multiply two numbers, introduced earlier in the course:

1. Draw a box labelled *x* and write the first input in it;
2. Draw a box labelled *y* and write the second input in it;
3. Draw a box labelled *result* and write 0 in it;
4. If the contents of *y* is 0 go to step 8;
5. Add the contents of box *x* to the contents of box *result*;
6. Subtract one from the contents of box *y*;
7. Go to step 4;
8. Finish.



What happens if the program is given inputs 6 and 2.5?

[This illustrates the importance of checking that inputs are as expected.]

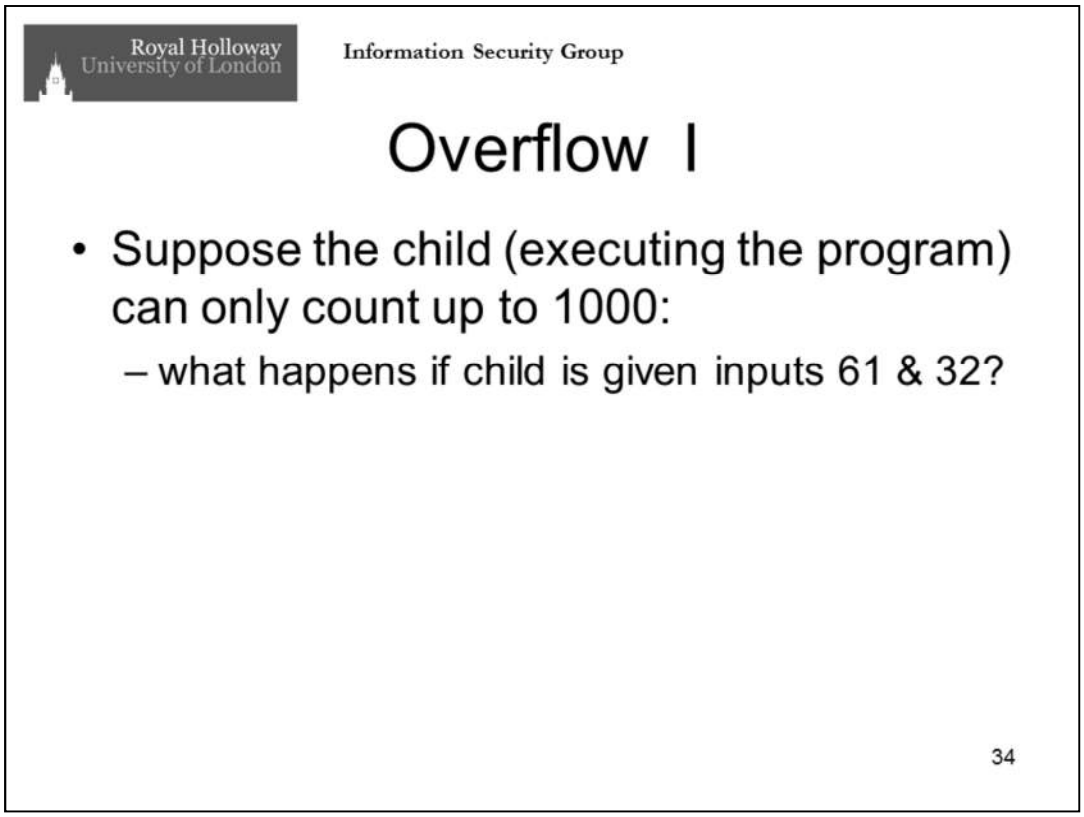


What happens if the program is given inputs 6 and 2.5?

As a result:

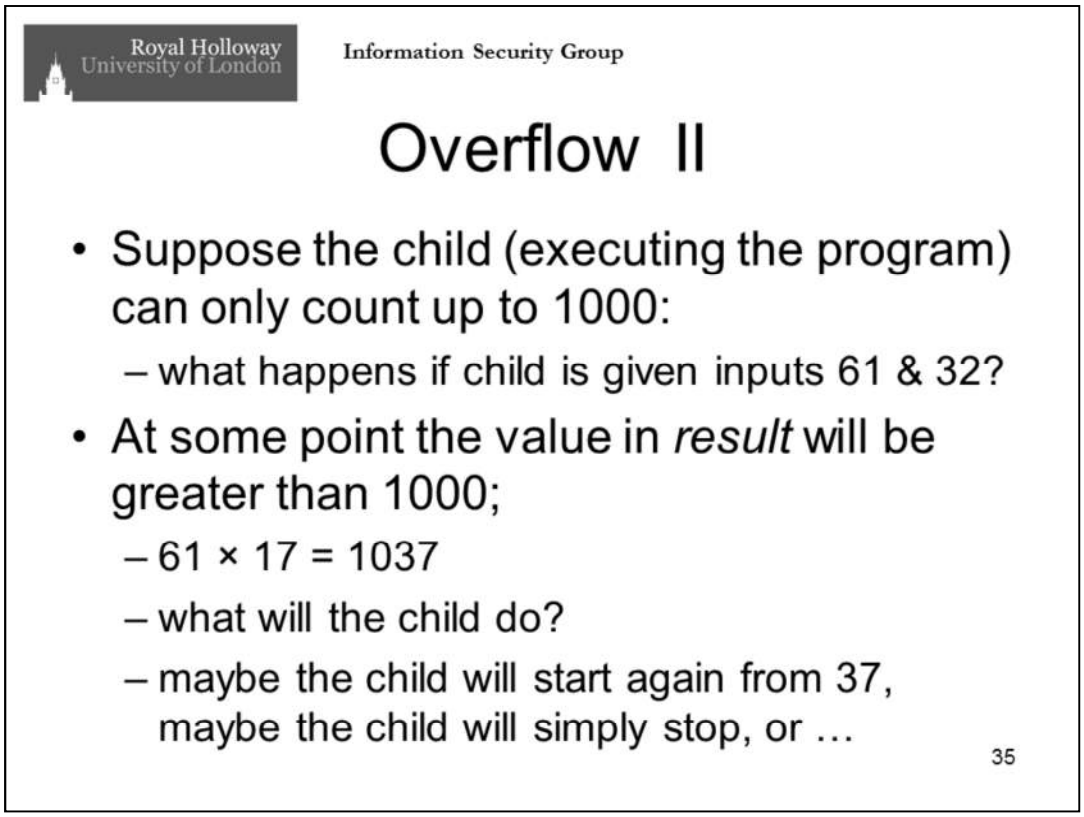
- the termination condition ($y = 0$) is never satisfied;
- the program enters an infinite loop.

That is, the program was written under the assumption that the inputs would be of the expected type. If this is not true, then the program can fail in unexpected and potentially dangerous ways.



Suppose the child (executing the program) can only count up to 1000:

- what happens if the child is given input 61 and 32?



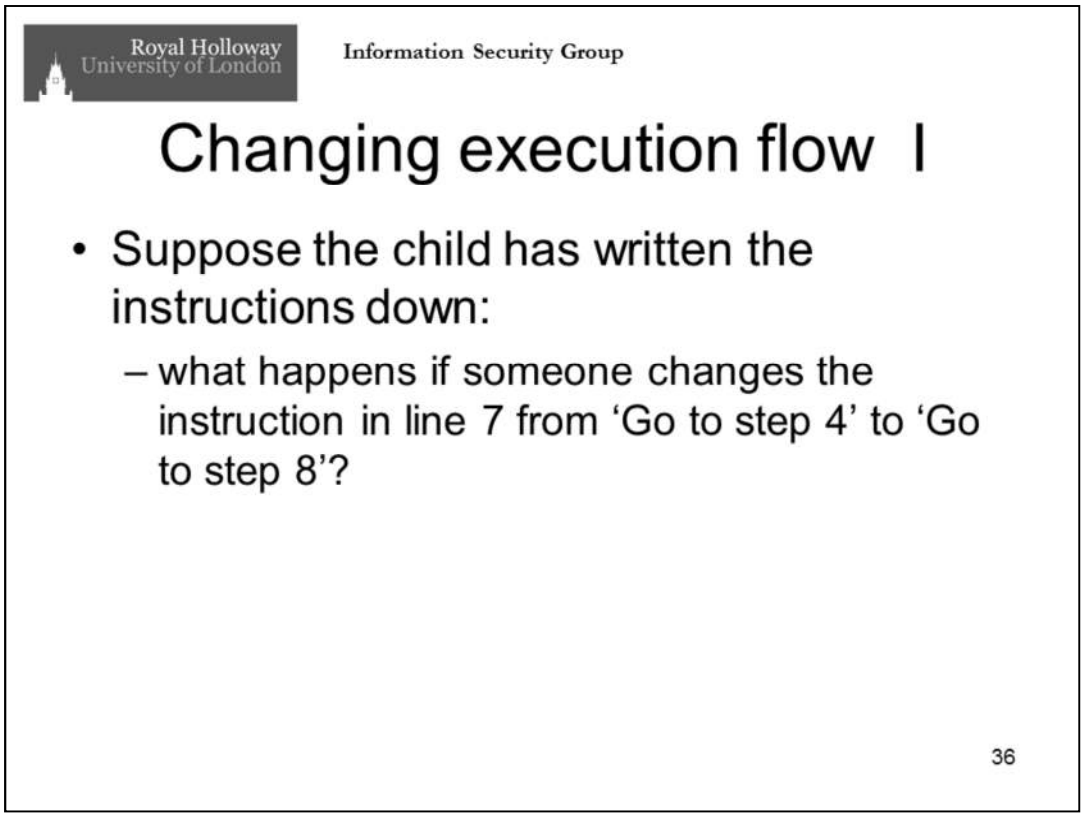
Suppose the child (executing the program) can only count up to 1000:

- what happens if the child is given input 61 and 32?

During execution, at some point the value in *result* will be greater than 1000;

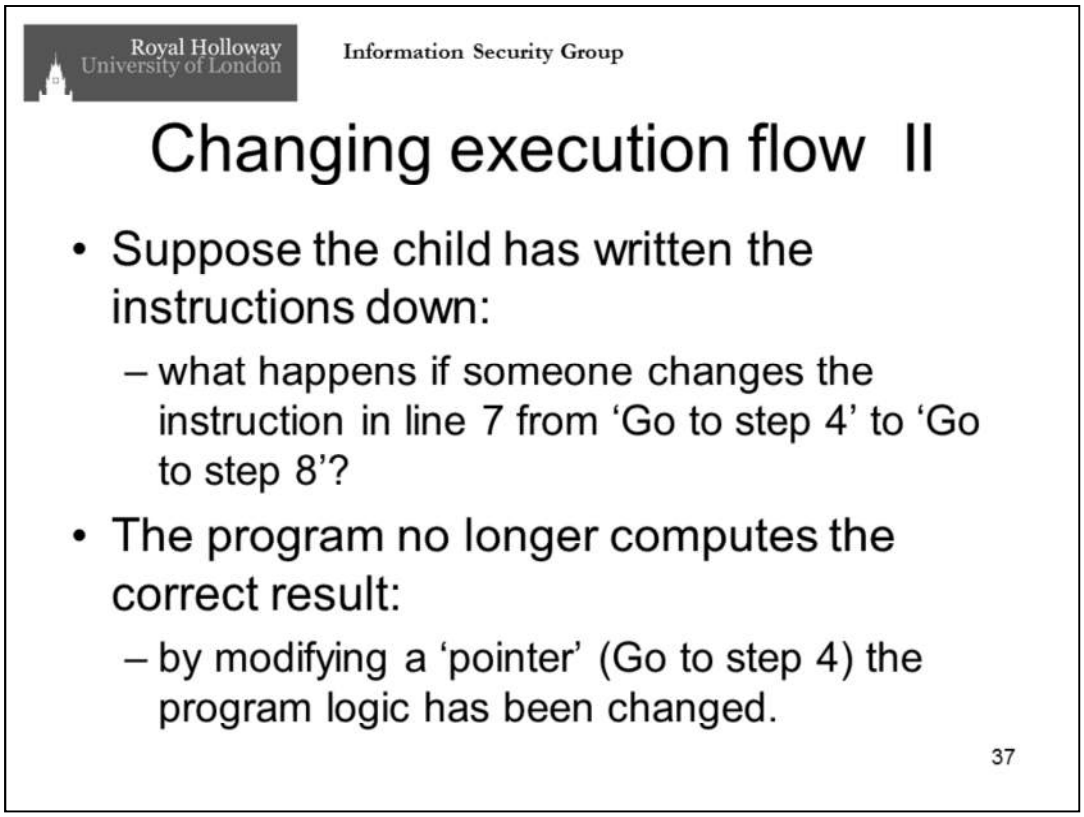
- 61 × 17 = 1037
- what will the child do?
- maybe the child will start again from 37, maybe the child will simply stop (generate an exception), …

In any event, the result will be not what is expected.



Suppose the child has written the instructions down:

• what happens if someone changes the instruction in line 7 from 'Go to step 4' to 'Go to step 8'?



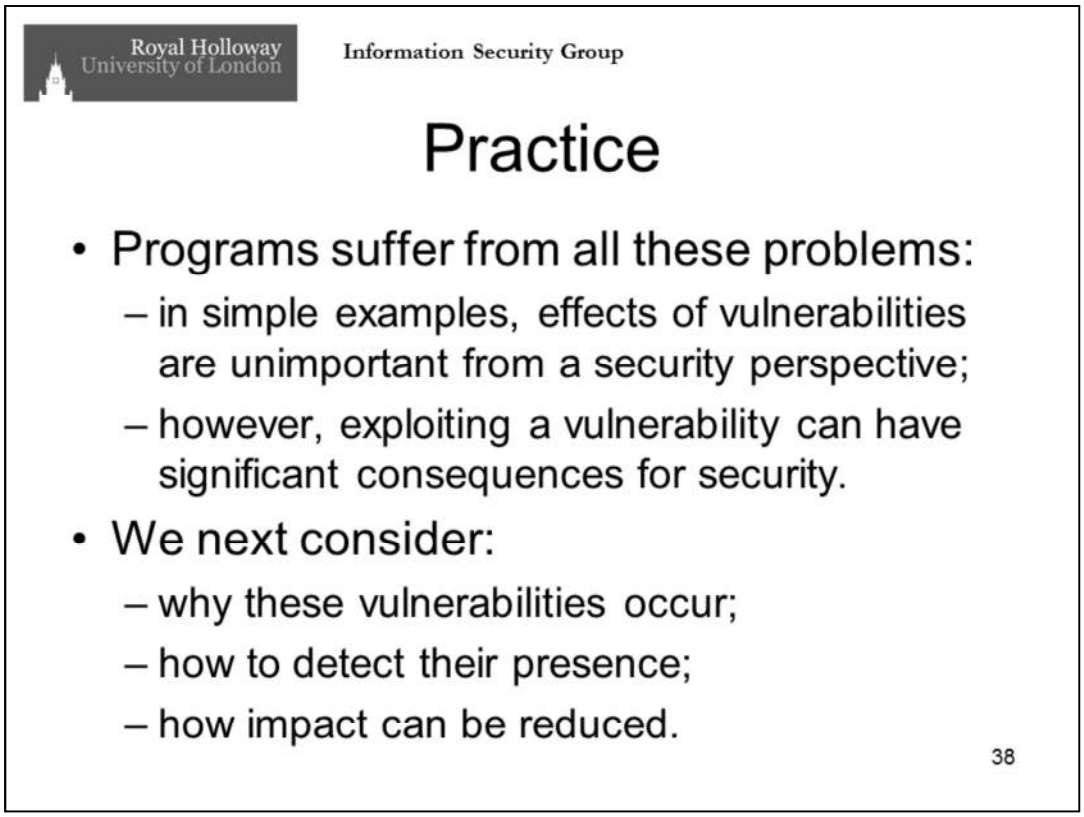
Suppose the child has written the instructions down:

• what happens if someone changes the instruction in line 7 from 'Go to step 4' to 'Go to step 8'?

The program no longer computes the correct result:

• by modifying a 'pointer' (Go to step 4) the program logic has been changed.

This underlines the importance of ensuring that the integrity of the code of the program must be guaranteed.



Computer programs suffer from all these types of vulnerabilities:

• in these simple examples, the effects of 'exploiting' these vulnerabilities are not important from a security perspective;

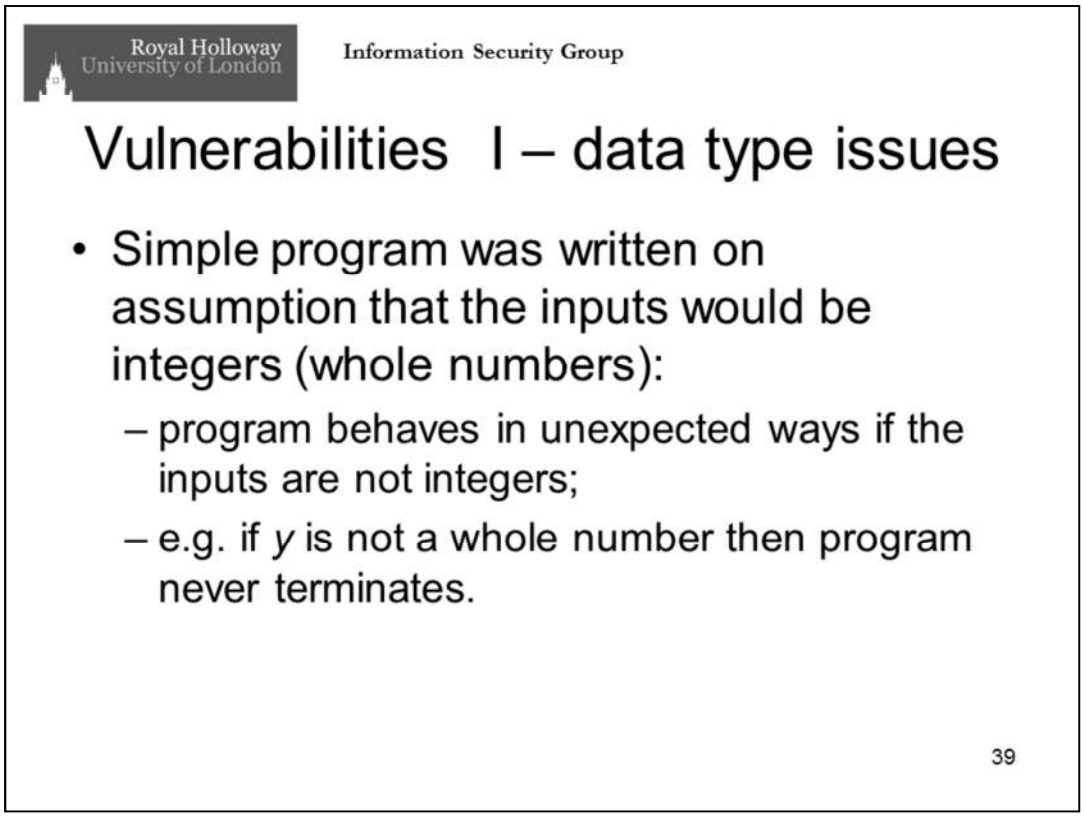
• however, in many situations exploiting a vulnerability can have significant consequences for security.

We next consider:

• why these vulnerabilities occur;

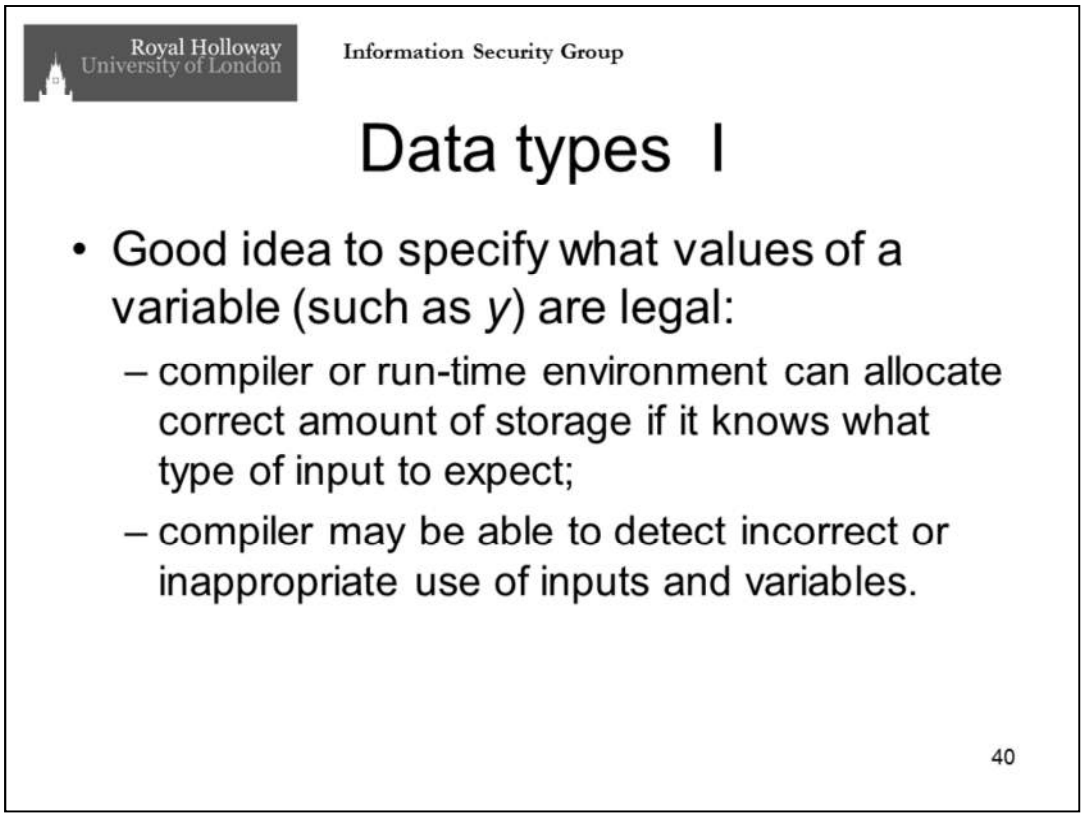
• how to detect the existence of such vulnerabilities;

• how the impact of these vulnerabilities can be reduced (i.e. we look at mitigations).



The simple program was written on assumption that the inputs would be integers (whole numbers). As a result, the program behaves in unexpected ways if the inputs are not integers.

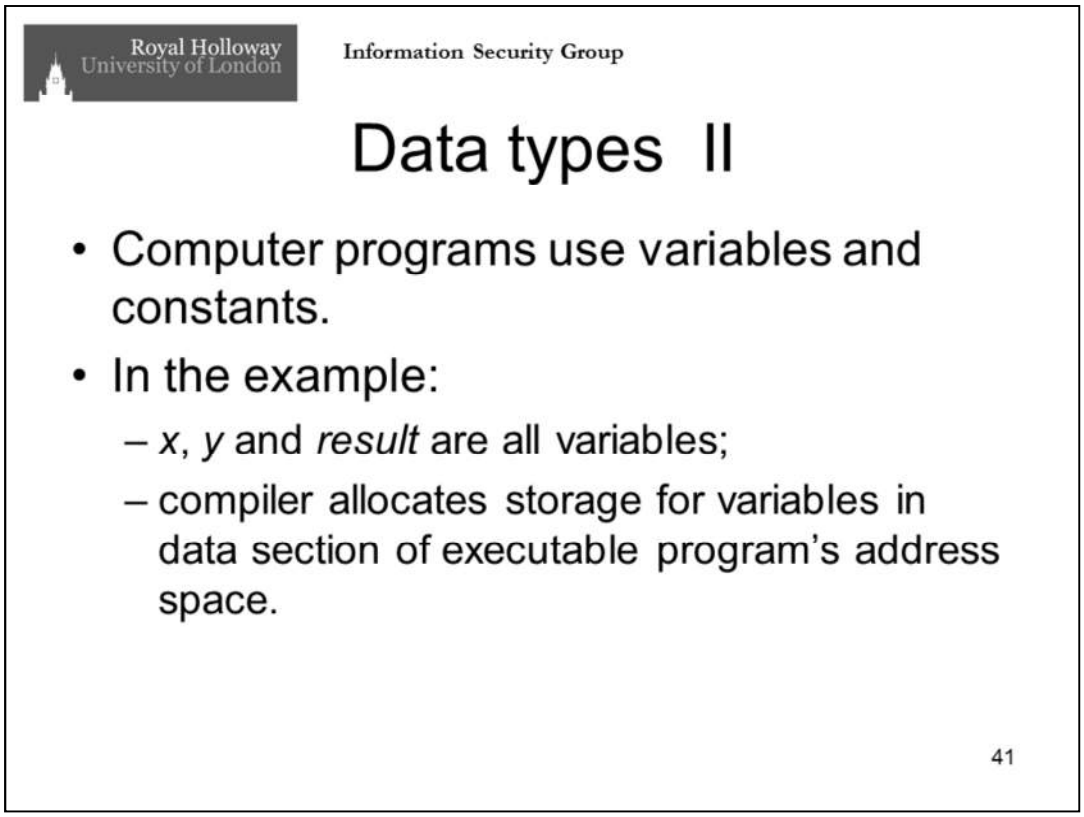
- for example, if *y* is not a whole number then the program never terminates.



It is generally a good idea to specify what values of a variable (such as *y*) are legal:

• the compiler or run-time environment can allocate an appropriate amount of storage if it knows what type of input to expect;

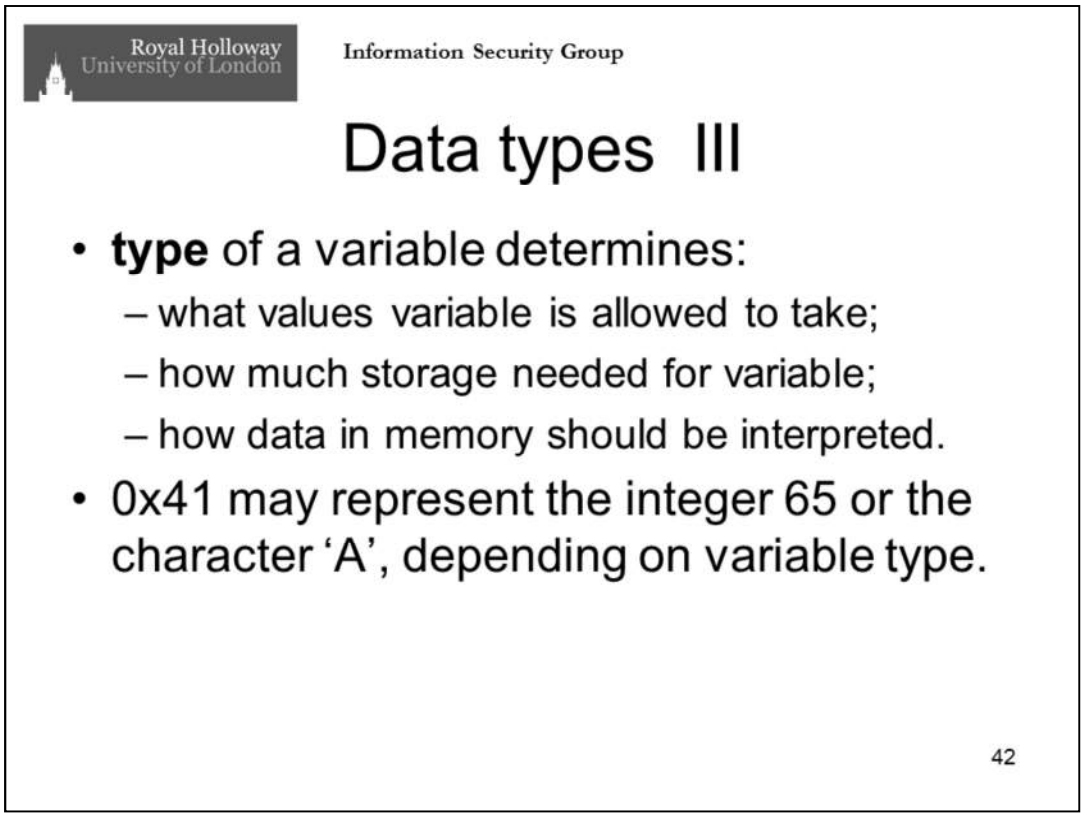
• the compiler may be able to detect incorrect or inappropriate use of inputs and variables.



Computer programs define variables and constants. In the simple example:

• *x*, *y* and *result* are all variables;

• the compiler allocates storage for variables in the data section of the executable program's address space.

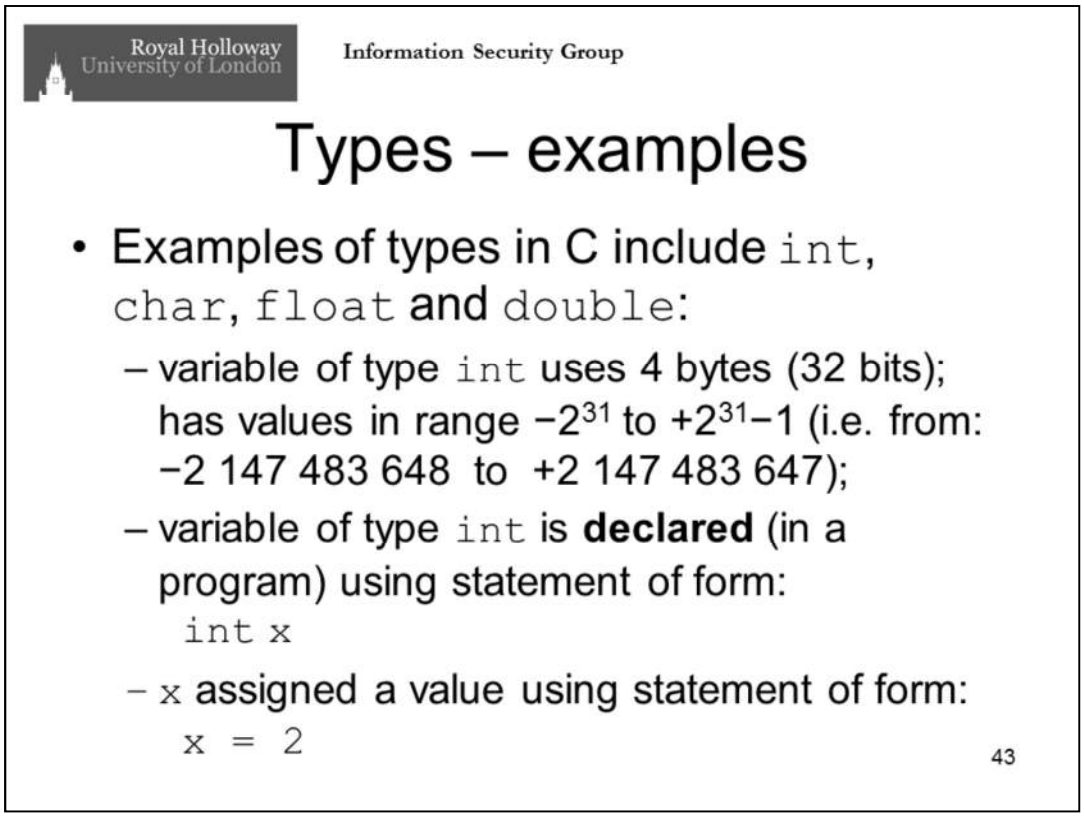


The **type** of a variable determines:

- what values the variable is allowed to take;
- how much storage is required for the variable;
- how a sequence of bits in memory should be interpreted.

For example, the sequence of bits 0x41 may represent the integer 65 (0x41 is 65 in decimal) or the character 'A' (0x41 is ASCII for 'A') depending on the type of the variable in which the value is stored.

ASCII (for American Standard Code for Information Interchange) is a standard way of encoding characters as strings of bits. ASCII is very widely used. The international standard for this encoding is ISO/IEC 646.

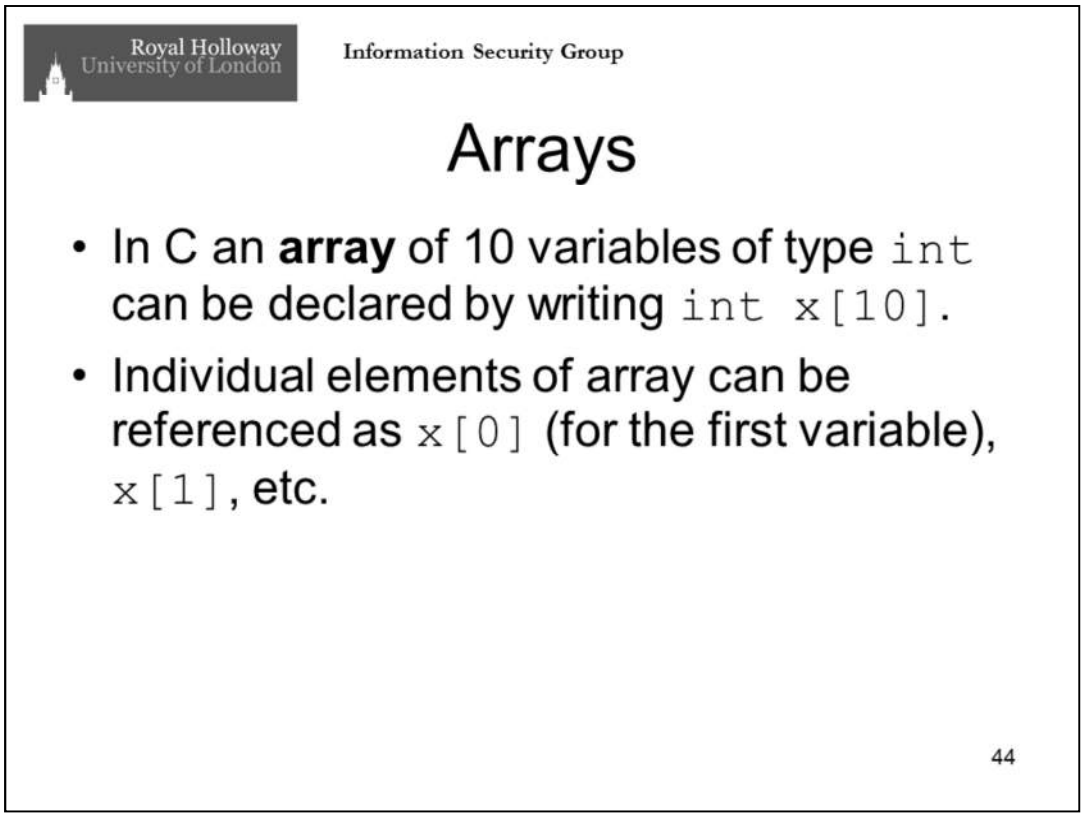


Examples of variable types (or **data types**) in the C programming language include `int`, `char`, `float` and `double`:

• a variable of type `int` requires 4 bytes (32 bits) of storage, and takes values in the range -2^{31} to $2^{31} - 1$ (i.e. from: −2 147 483 648 to +2 147 483 647);

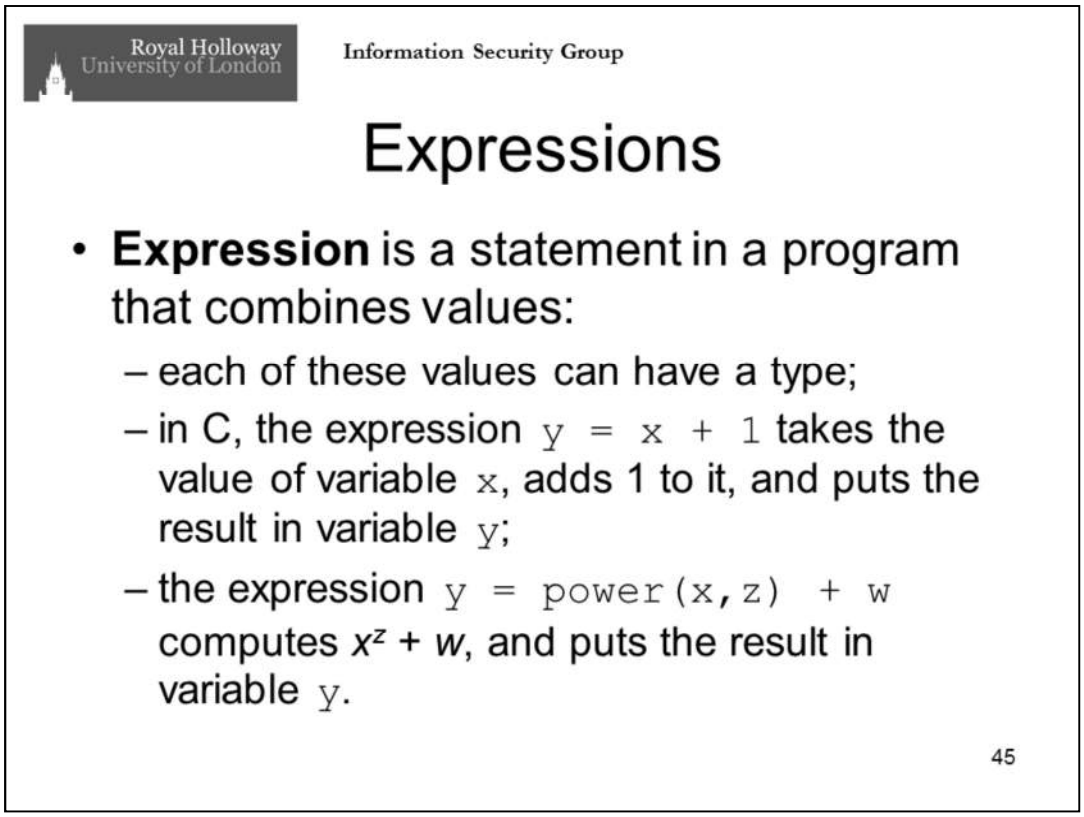
• a variable of type `int` is **declared** (in a program) using a statement of the form: `int x`

• in the body of a program, `x` is assigned a value using a statement of the form: `x = 2`



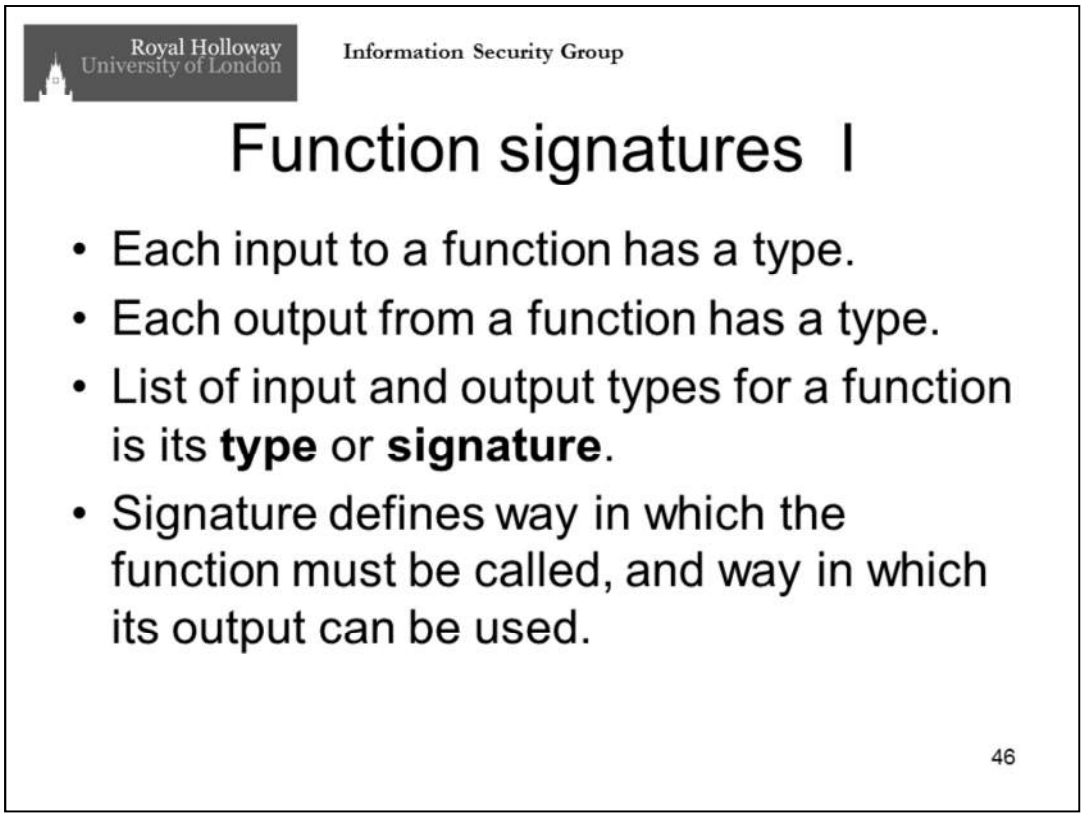
In the C language, an array of 10 variables of type `int` can be declared by writing `int x[10]`.

Individual elements of the array can be referenced as `x[0]` (for the first variable), `x[1]`, etc., up to `x[9]`.



An **expression** is a statement in a program that specifies a way of combining values:

- each of these values can have a type;
- in C, the expression `y = x + 1` takes the value of variable `x`, adds 1 to it, and puts the result in variable `y`;
- the expression `y = power(x,z) + w` computes $x^z + w$, and puts the result in variable `y`.

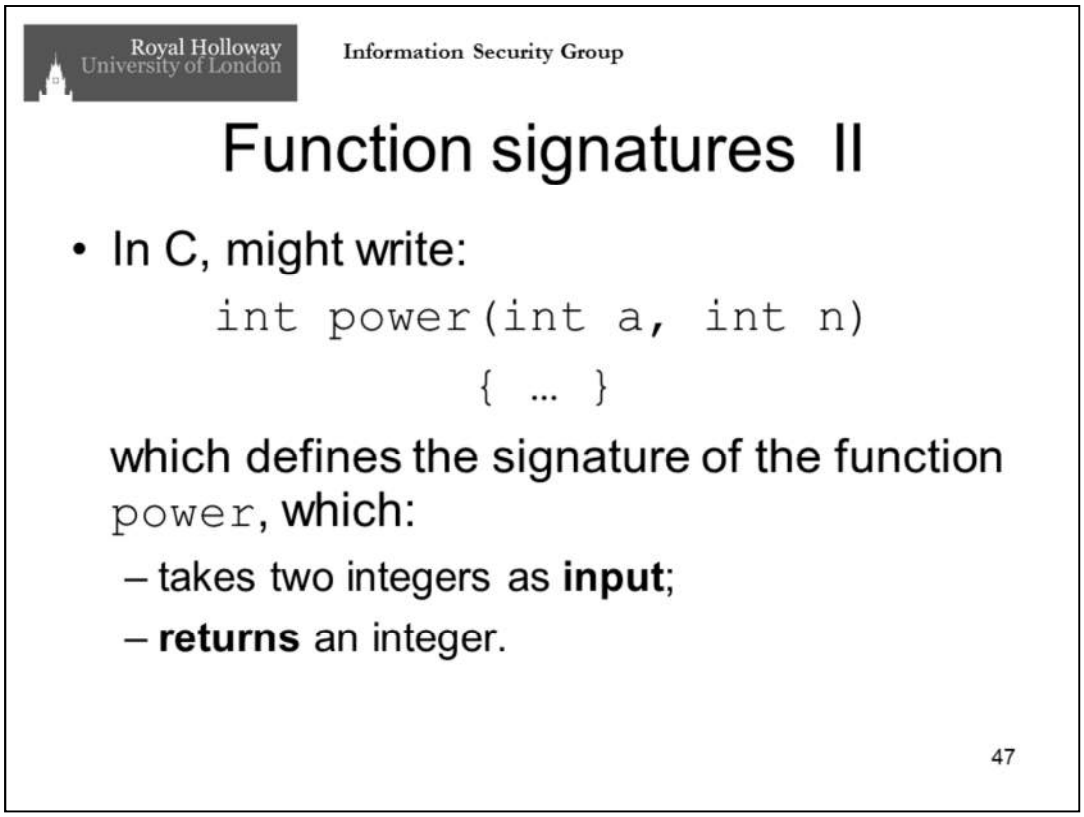


Each input to a function has a type, and each output from a function has a type.

The list of input and output types for a function is its **type** or **signature**.

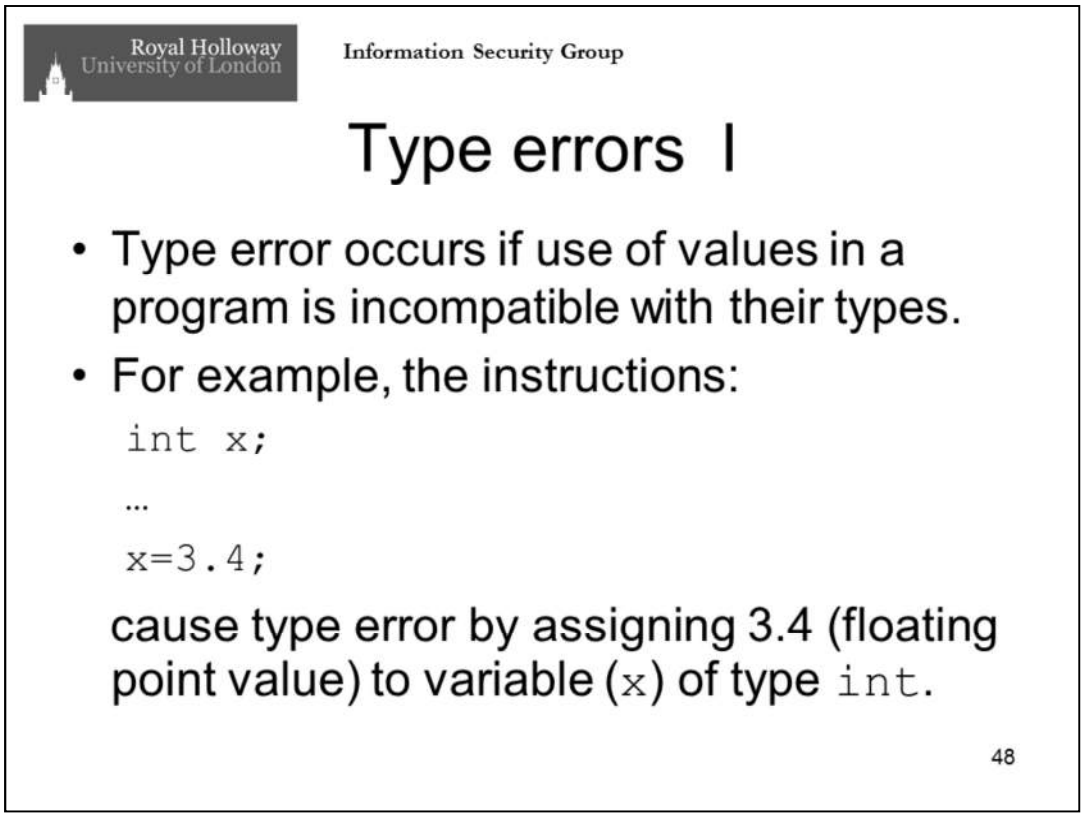
The signature defines the way in which the function must be called, and the way in which its output can be used.

If you think of a function as a 'black box', the function signature defines the nature of what the black box takes in and what it gives out.



In C, we might write: `int power(int a, int n) { … }` which defines the signature of the function `power`. This function:

- takes two integers as **input**;
- **returns** an integer.

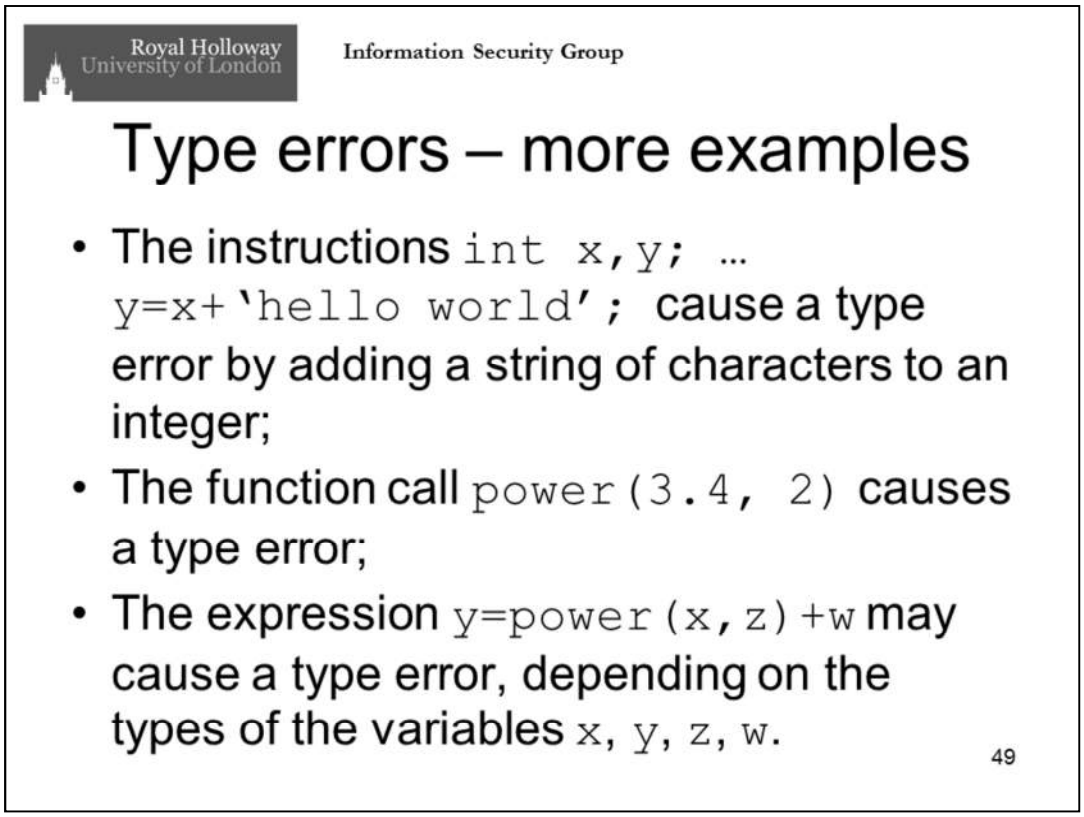


A type error occurs if the use of values in a program is incompatible with the types of those values.

For example, the sequence of instructions:

```
int x;
 …
 x=3.4;
```

cause a type error by assigning a floating point value (i.e. 3.4) to a variable (i.e. `x`) of type `int`.



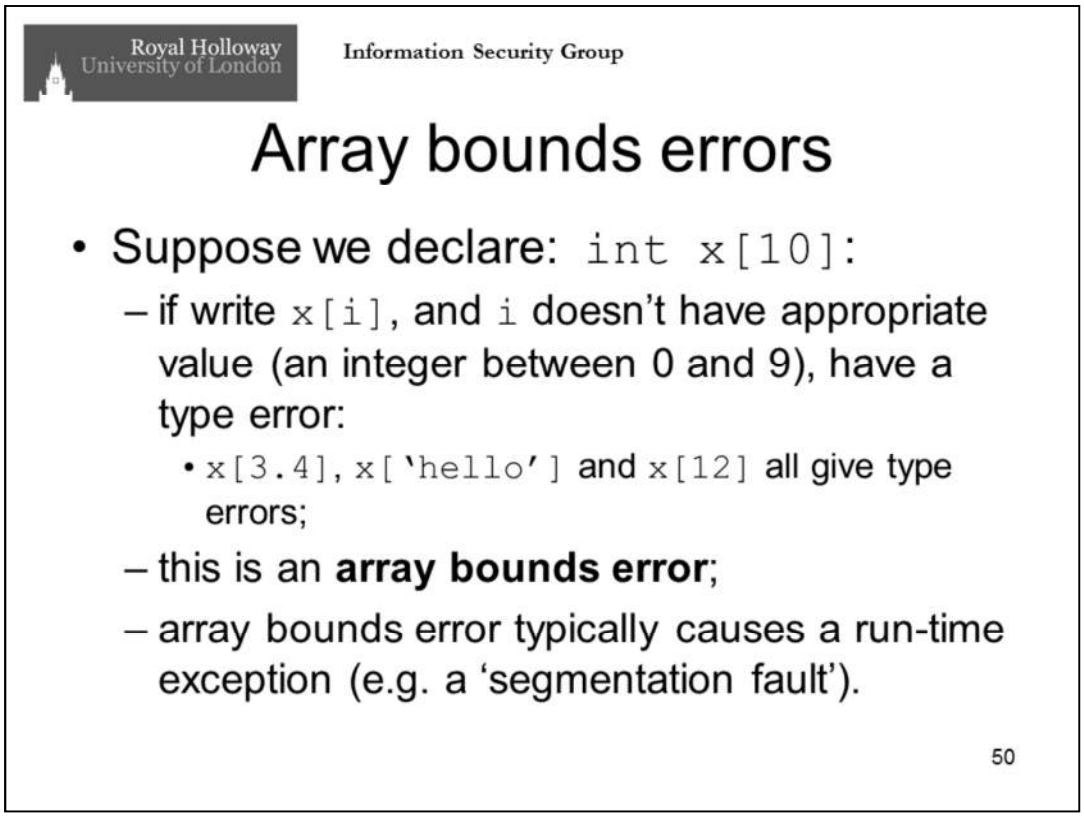
The sequence of instructions:

```
int x,y;
 ...;
 y=x+'hello world'
```

cause a type error by trying to add a string of characters to an integer;

The function call `power(3.4, 2)` causes a type error;

The expression `y=power(x,z)+w` may cause a type error, depending on the types of the variables `x`, `y`, `z`, `w`.



Suppose we declare an array as follows:

```
int x[10]
```

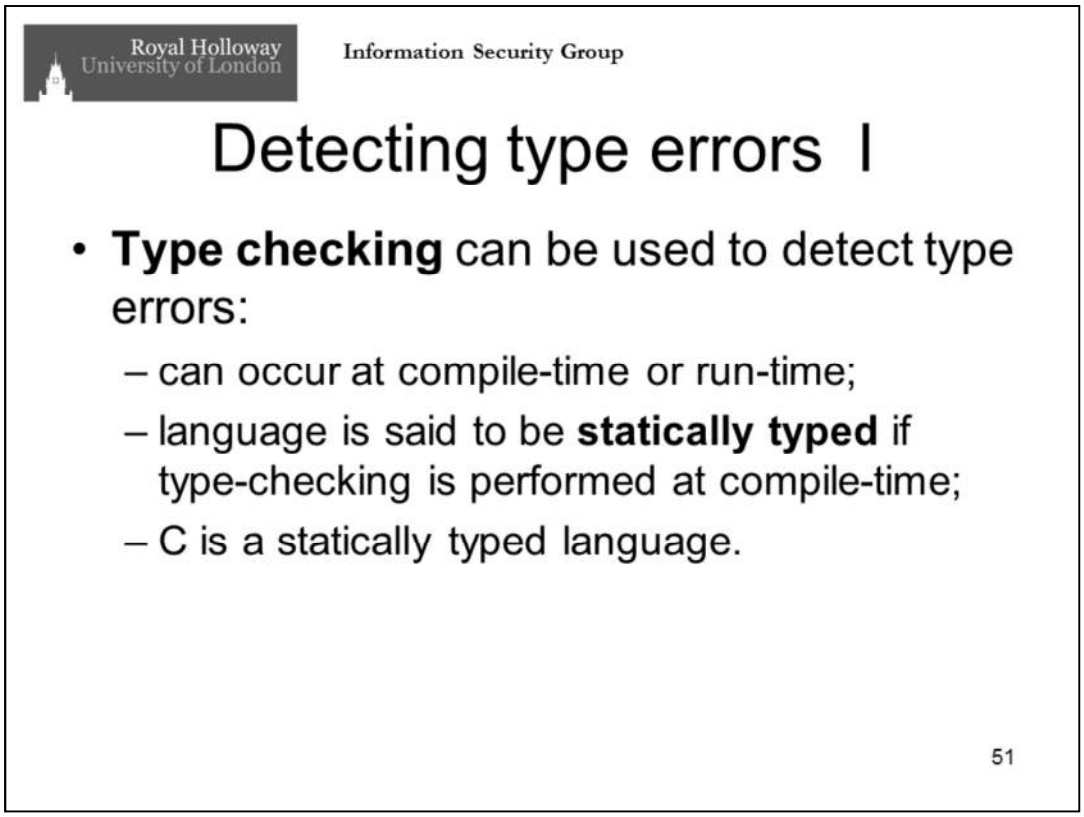
Then:

• if we write `x[i]` and `i` does not have an appropriate value (an integer between 0 and 9 inclusive), then we have a type error. For example:

- `x[3.4]`, `x['hello']` and `x[12]` will all cause type errors;

• a type error of the form `x[12]` is an array bounds error;

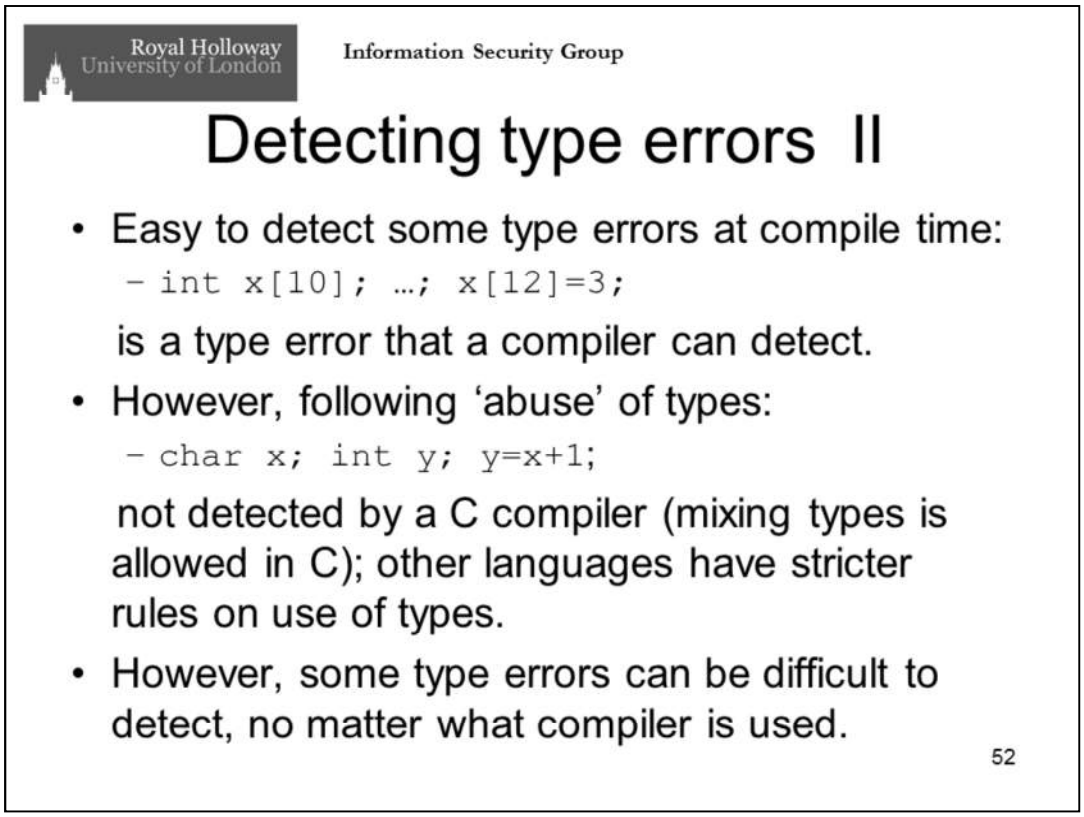
• an array bounds error typically causes a run-time exception (e.g. a 'segmentation fault').



A process known as **Type checking** can be used to detect type errors:

- it can occur at the time the program is compiled or when it is run;
- a language is said to be **statically typed** if type-checking is performed at compile-time;
- C is a statically typed language.

Type checking involves verifying that variables are used appropriately, i.e. in line with their declared types.



Some kinds of type errors can readily be detected at the time of compilation of a program. For example

```
int x[10]; x[12]=3;
```

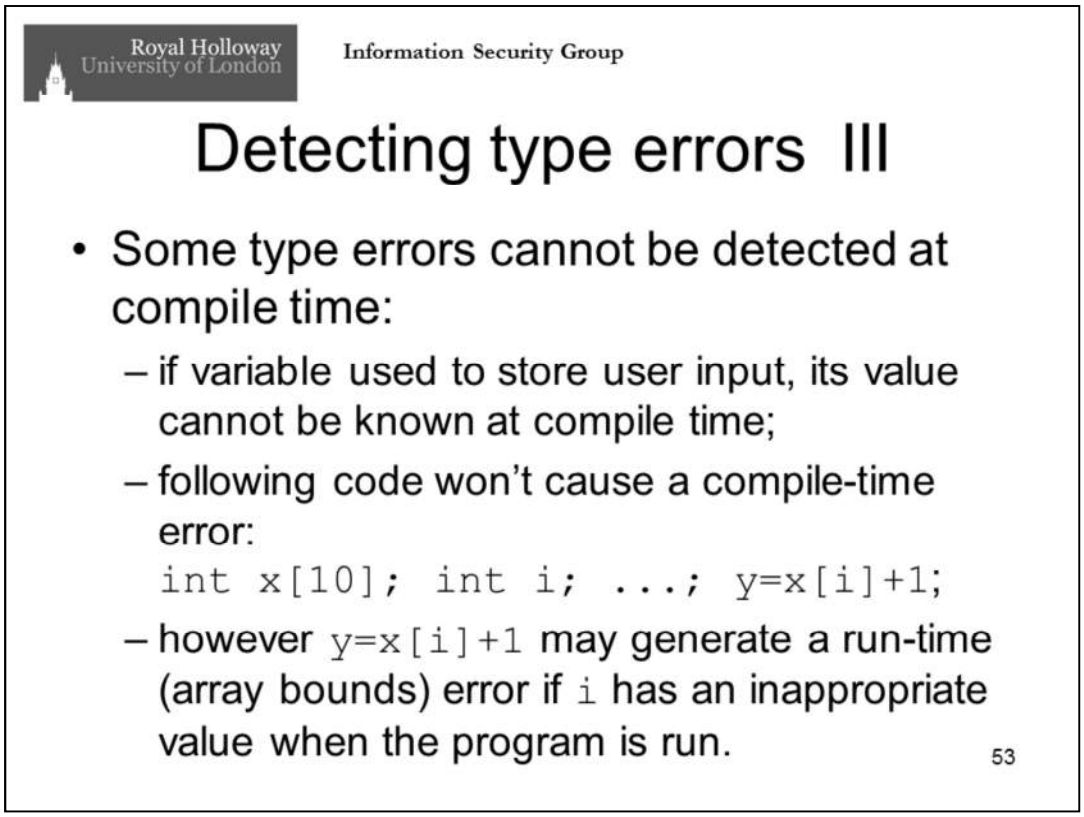
is a type error that can be detected by the compiler.

However, the following code:

```
char x; int y; y=x+1;
```

will not be detected by a standard c compiler (mixing types in this way is permitted in c), but this type of assignment would be detected by compilers for other languages which impose stricter conditions on use of types. Of course such abuse of types is potentially unsafe, since, amongst other things, the result will depend on how the compiler converts characters to bit strings.

However, some type errors can be difficult to detect, no matter what compiler is used.

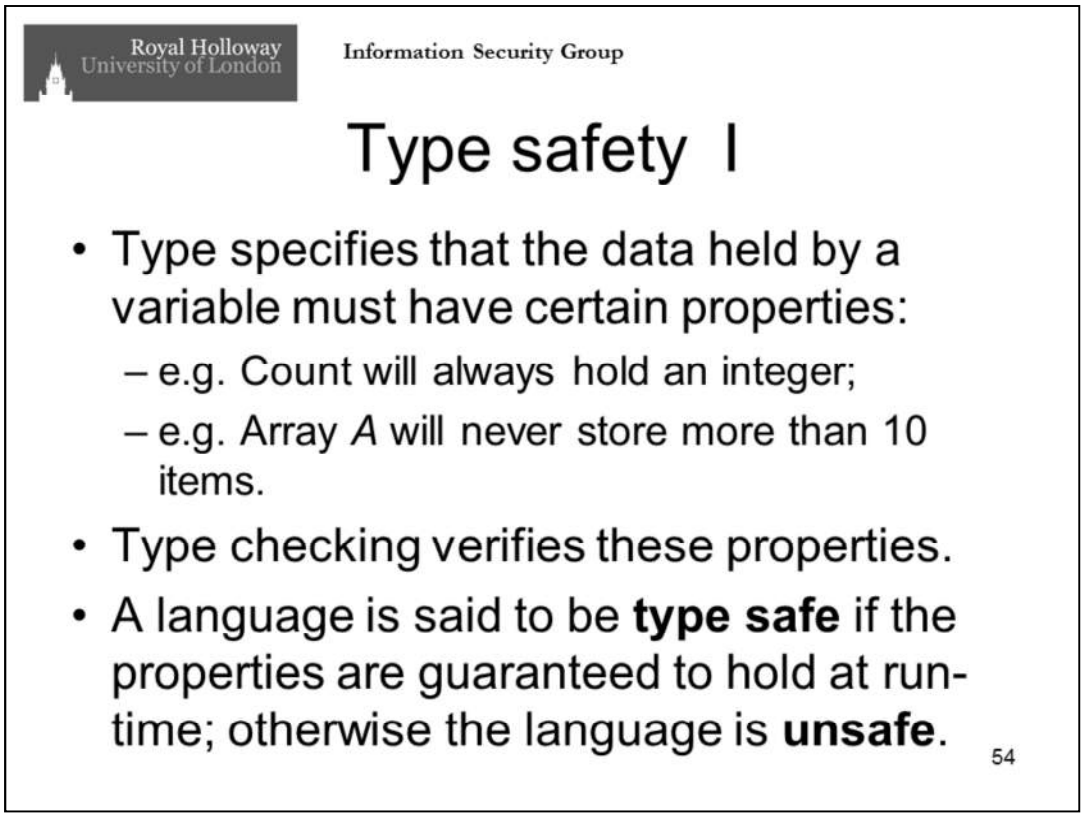


Some type errors cannot be detected at compile time. If a variable is used to store user input, its value cannot be known at compile time.

For example, the following code will not generate a compile-time error:

```
int x[10];
int i;
...;
y=x[i]+1;
```

However the expression `y=x[i]+1` may generate a run-time (array bounds) error if `i` has an inappropriate value when the program is run. Whether or not a run-time error is generated by this type of array bounds problem will depend on what checking is performed whilst a program is executing.

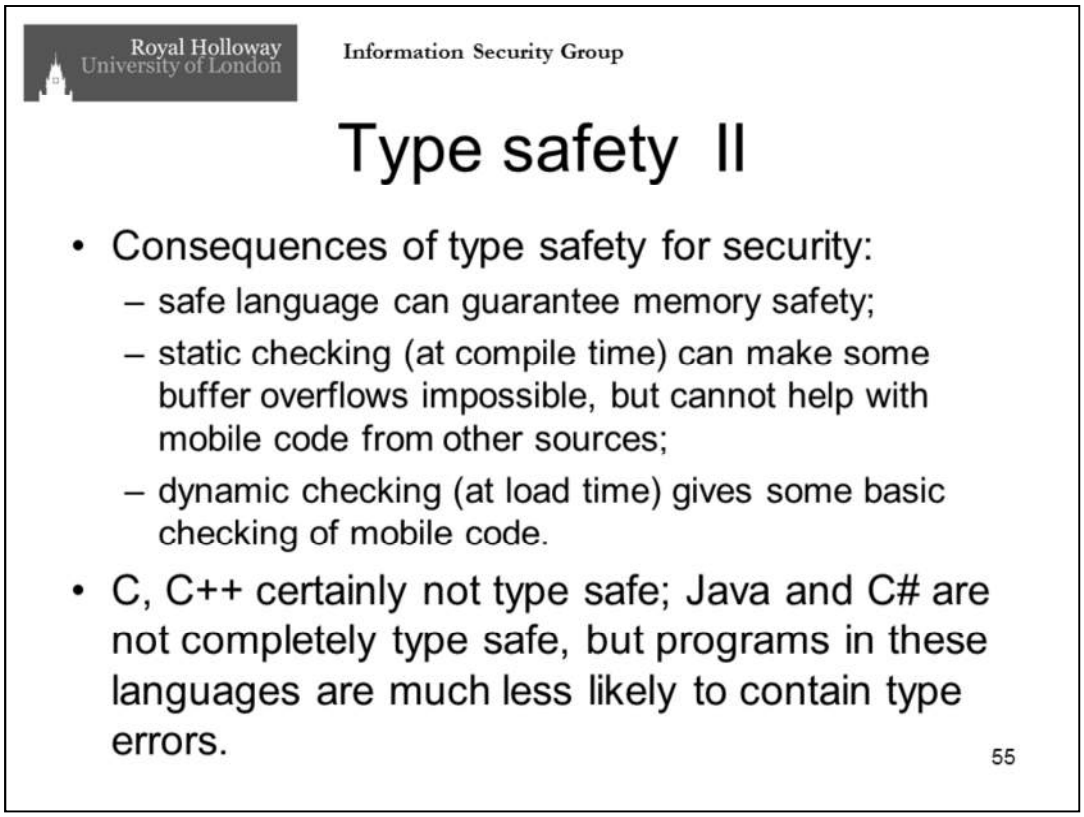


As we have discussed, the type of a variable specify that the data held by that variable must have certain properties:

- e.g. Count will always hold an integer;
- e.g. Array *A* will never store more than 10 items.

Type checking involves verifying these properties, either at compile-time or at run-time.

A language is said to be **type safe** if the type properties are guaranteed to hold at run-time; otherwise the language is said to be **unsafe**.

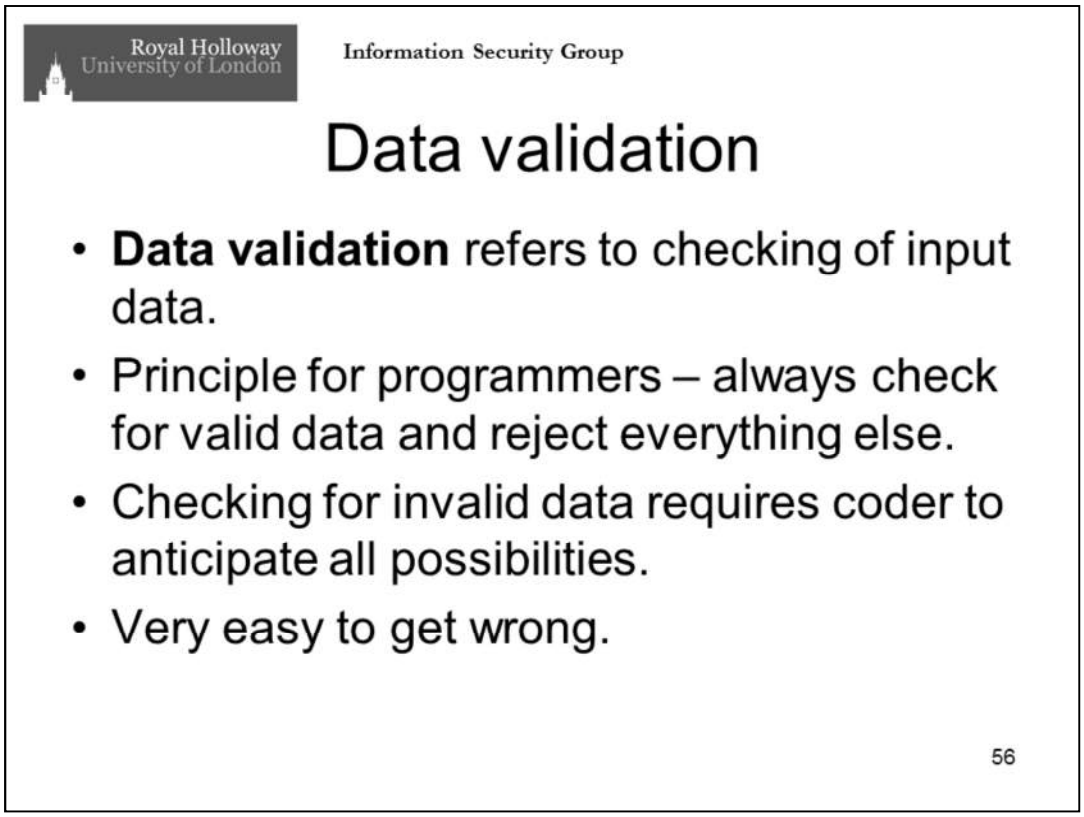


Consequences for security of type safety:

- a safe language can guarantee memory safety;
- static checking (at compile time) can make some buffer overflows impossible, but cannot help with mobile code from other sources;
- dynamic checking (at load time) gives some basic checking of mobile code.

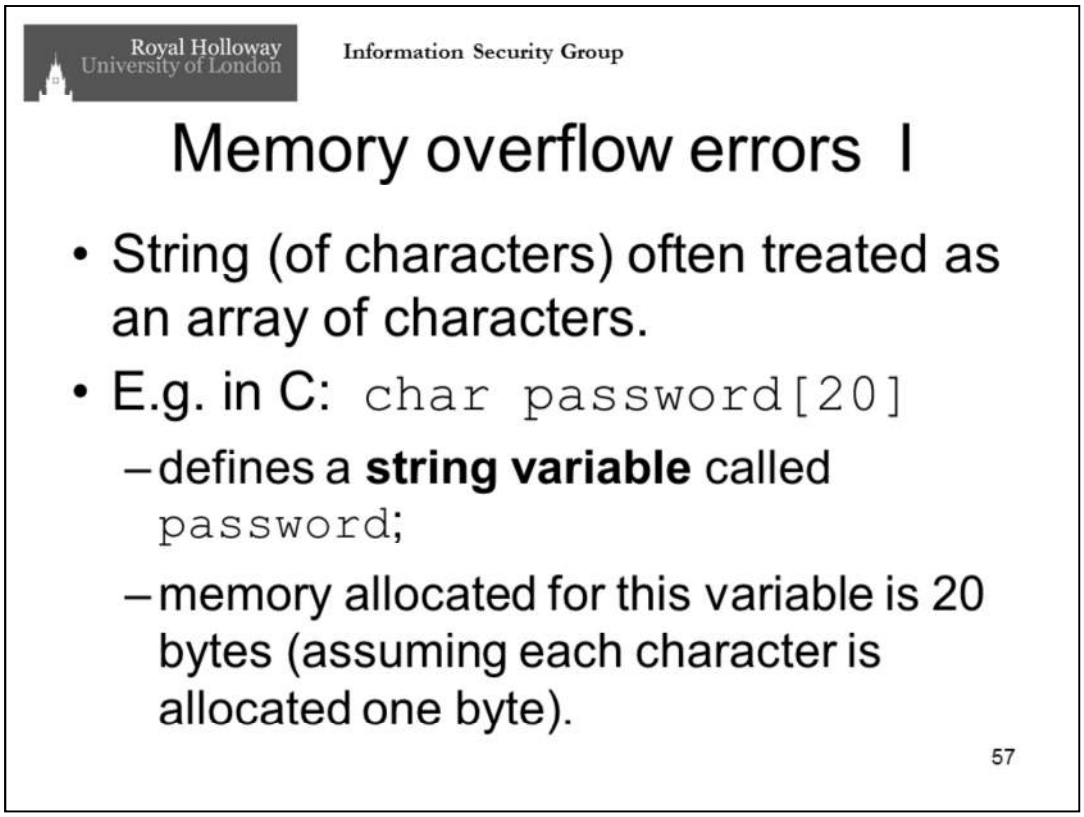
C and C++ are certainly not type safe; Java and C# are also not completely type safe, but programs written in these languages are very much less likely to contain type errors.

C# can be made type safe by prohibiting untyped pointers. Use of such pointers can be prohibited at the compiler level.



Data validation refers to the checking of input data. Always checking for valid data and rejecting everything else is a fundamentally important principle for programmers.

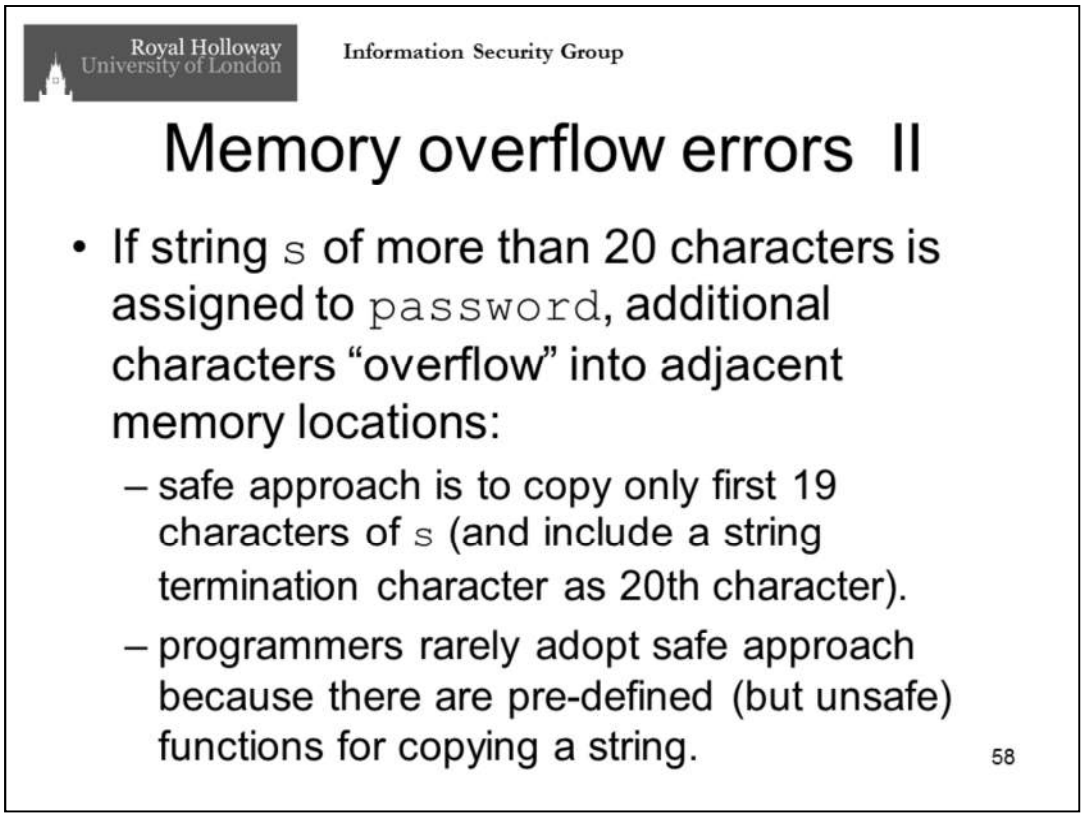
Checking for invalid data requires the coder to anticipate all possibilities. This is very easy to get wrong.



A string (of characters) is often treated as an array of characters. For example, in C we might write:

```
char password[20]
```

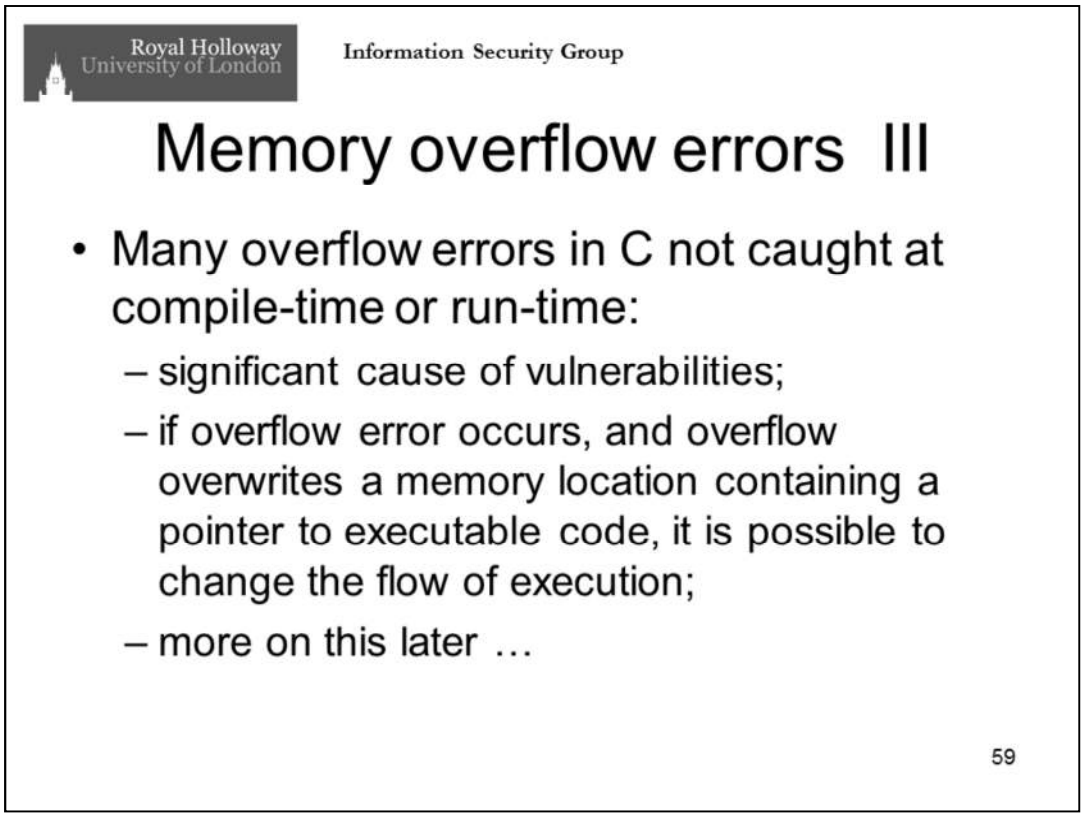
- this defines a **string variable** called `password`;
- the memory allocated for this variable is 20 bytes (assuming each character is allocated a single byte).



If a string `s` of more than 20 characters is assigned to `password`, the additional characters may "overflow" into adjacent memory locations:

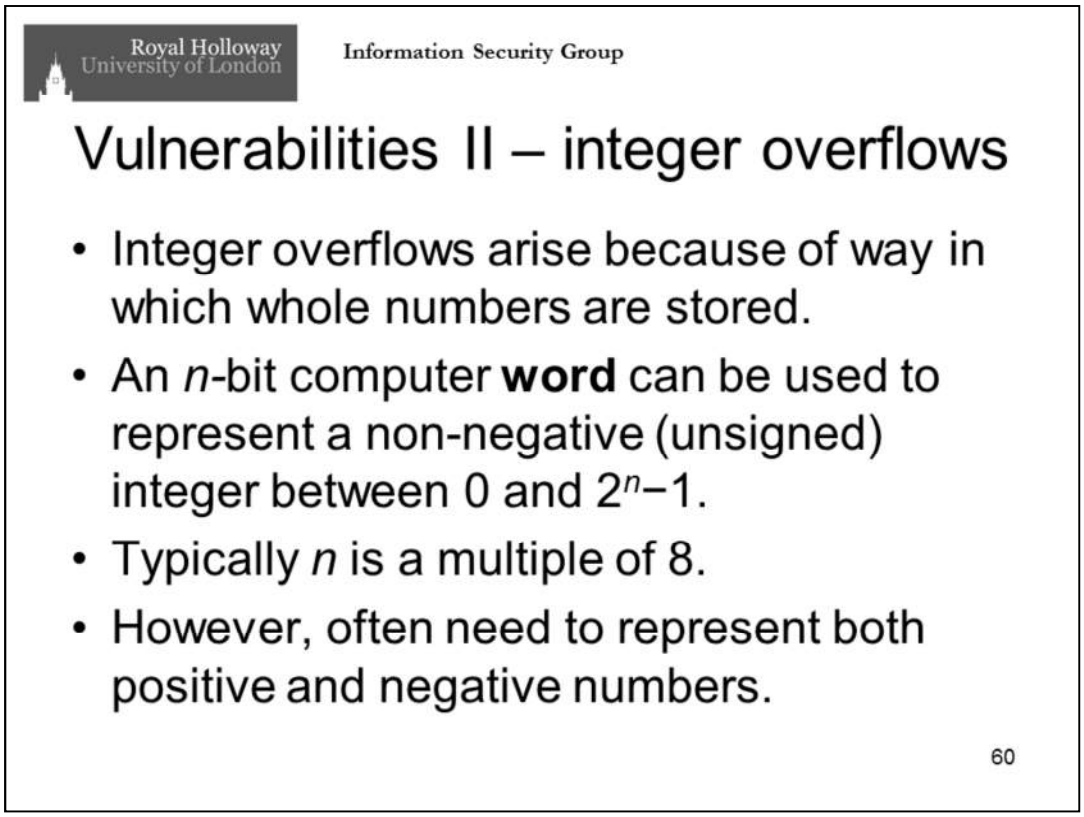
• the safe approach is to copy only the first 19 characters of `s` (and include a string termination character as the 20th character).

• programmers rarely adopt the safe approach because there are pre-defined (but unsafe) functions for copying a string.



Many overflow errors in C are not caught at compile-time or run-time. They are a significant cause of vulnerabilities.

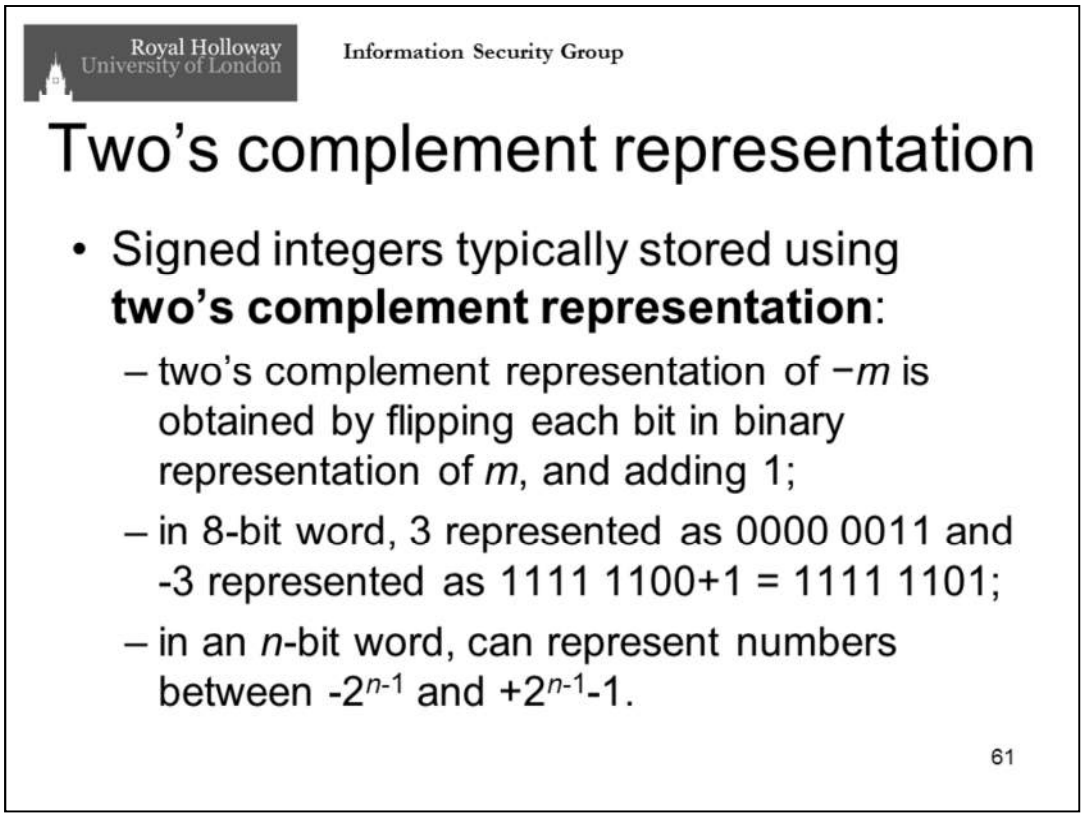
If an overflow error occurs and the overflow overwrites a memory location containing a pointer to executable code, it is possible to change the flow of execution. This is discussed in greater detail later …



Integer overflows arise because of the way in which whole numbers are stored in a computer.

An *n*-bit computer **word** can be used to represent a non-negative (unsigned) integer between 0 and 2^n-1, where typically *n* is a multiple of 8.

However, we often need to represent both positive and negative numbers.



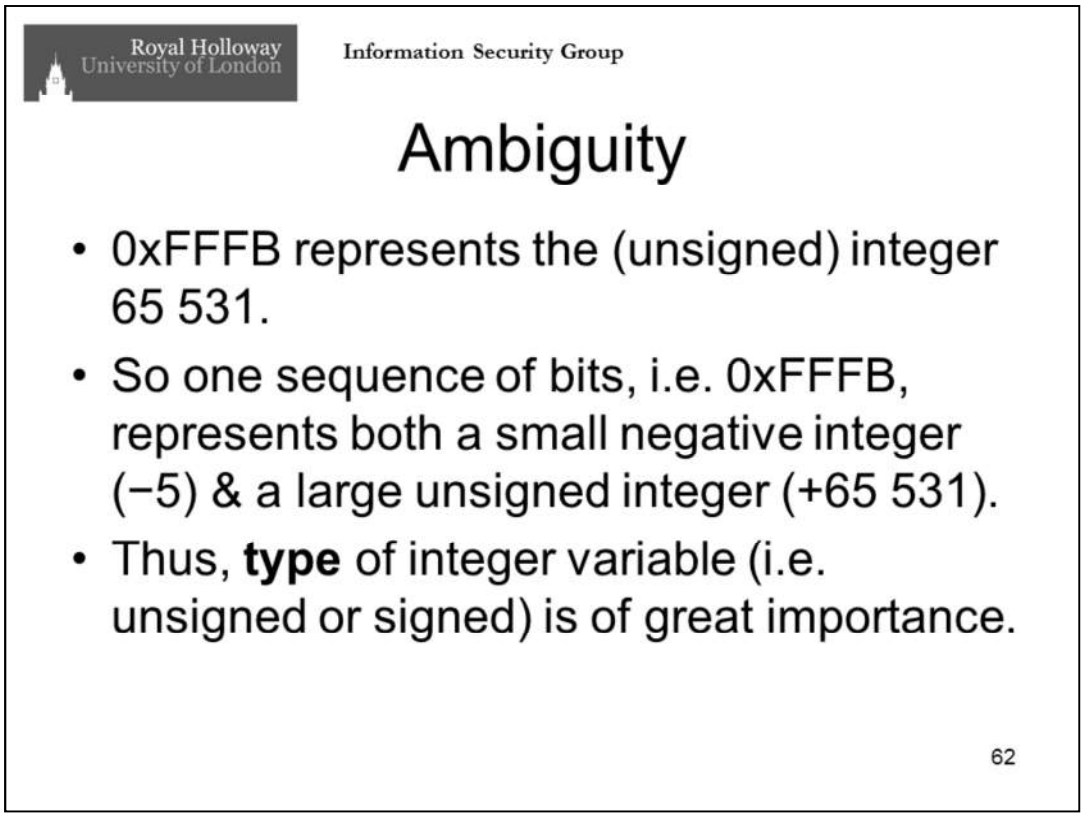
Signed integers are typically stored using the **two's complement** representation:

• the two's complement representation of −*m* is obtained by flipping each bit in the binary representation of *m* (i.e. changing every one to a zero and every zero to a one) and adding 1;

• in an 8-bit word, 3 is represented as 0000 0011 and -3 is represented as 1111 1100+1 = 1111 1101;

• in an *n*-bit word, using this representation it is possible to represent numbers between -2^{n-1} and $+2^{n-1}-1$.

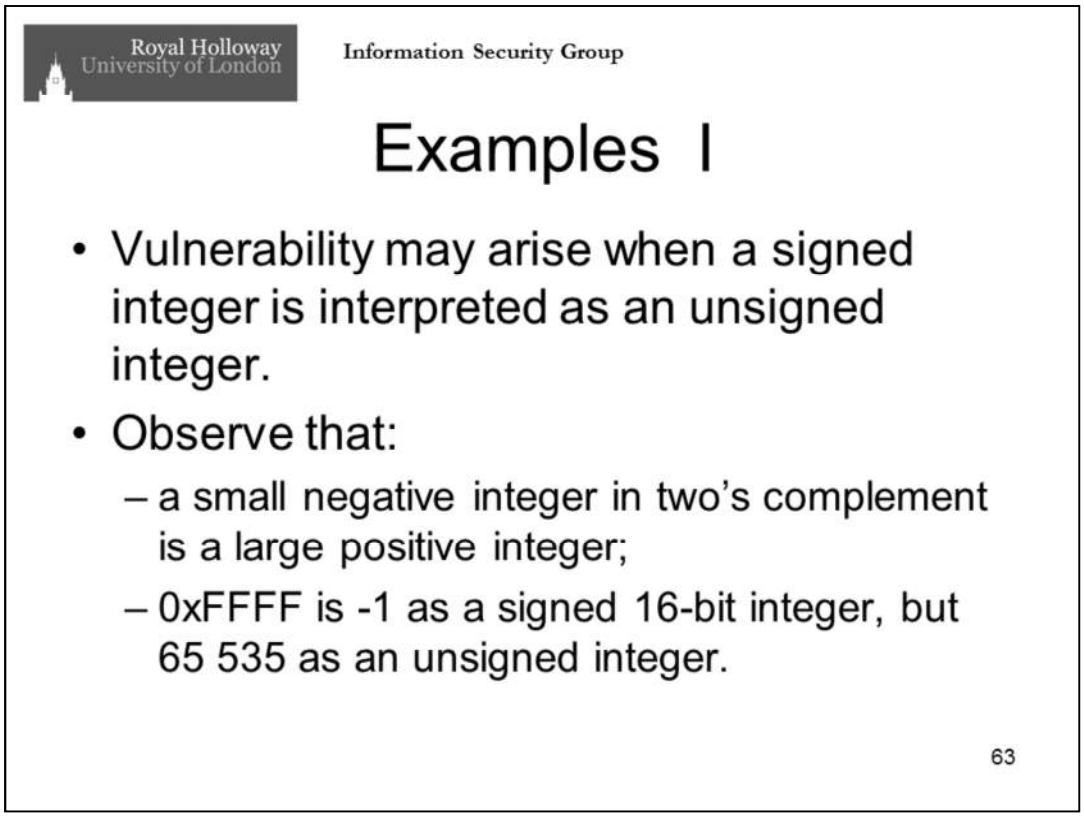
Note that, when using a two's complement representation, the most significant bit of a word is 1 for negative numbers and 0 for positive numbers (including zero).



0xFFFB represents the (unsigned) integer 65 531.

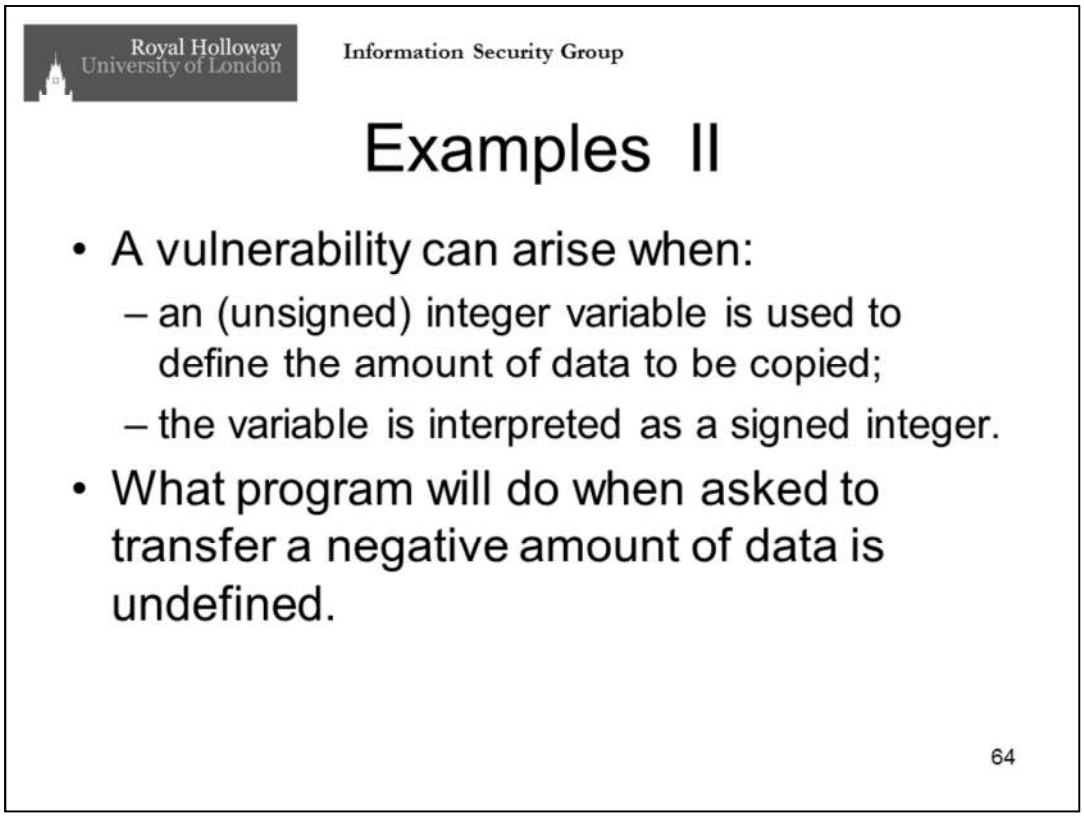
Hence the same sequence of bits, i.e. 0xFFFB, represents both a small negative integer (−5) and a large unsigned integer (+65 531).

In other words, the **type** of an integer variable (i.e. whether it is unsigned or signed) is of great importance.



A vulnerability may arise when a signed integer is implicitly converted to an unsigned integer. We observe that:

- a small negative integer in two's complement is a large positive integer;
- 0xFFFF is -1 as a signed 16-bit integer, but 65 535 as an unsigned integer.

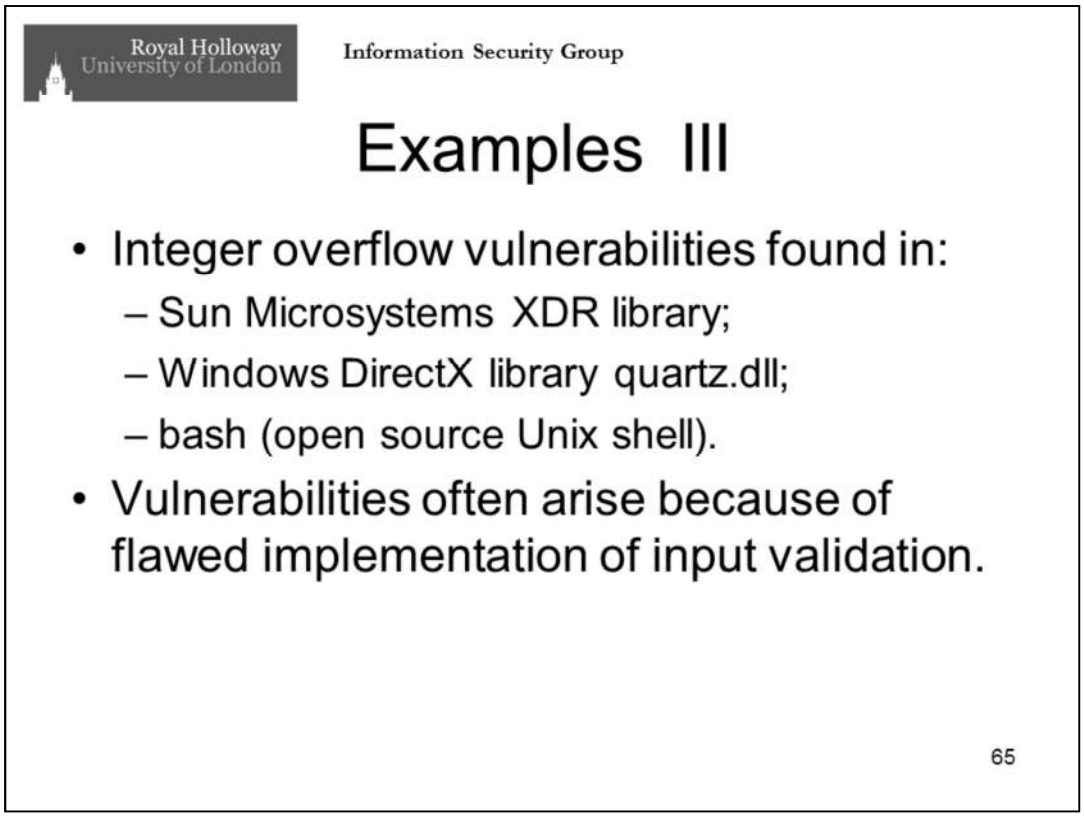


A vulnerability can arise when:

• an (unsigned) integer variable is used to define the amount of data to be copied;

• the variable is interpreted as a signed integer.

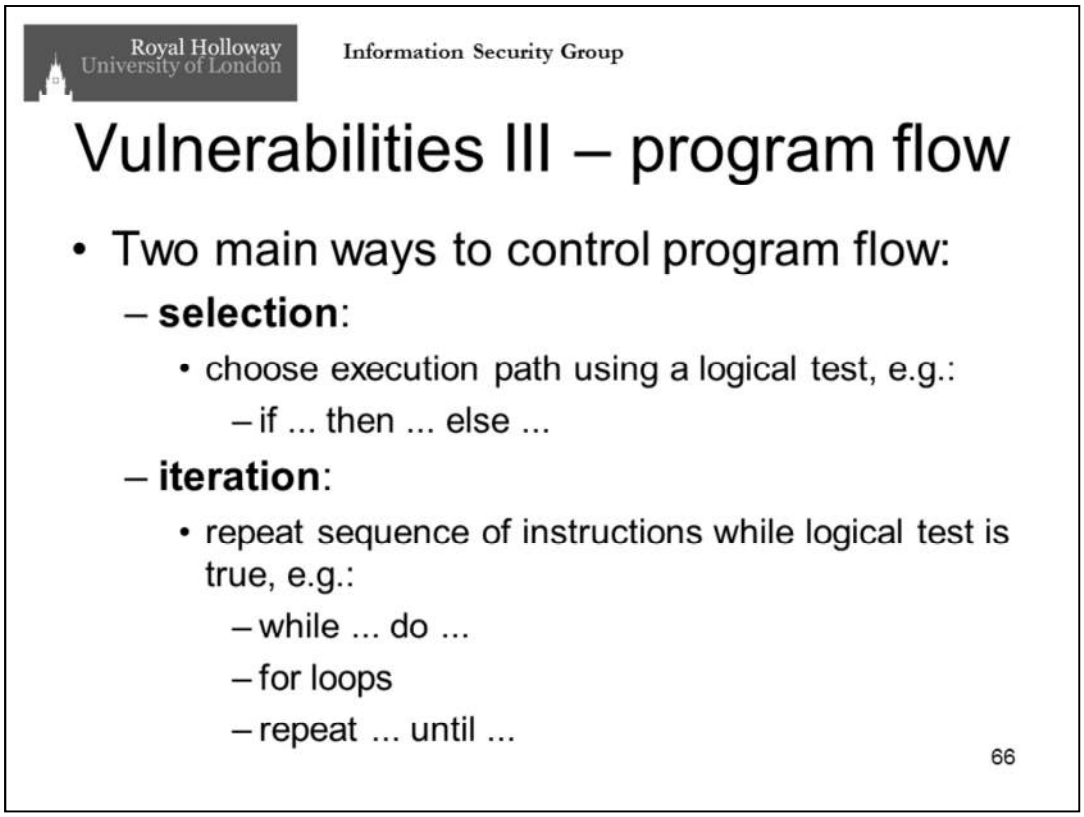
How the program will work when asked to transfer a negative amount of data is undefined.



Integer overflow vulnerabilities are widespread. They have been found in:

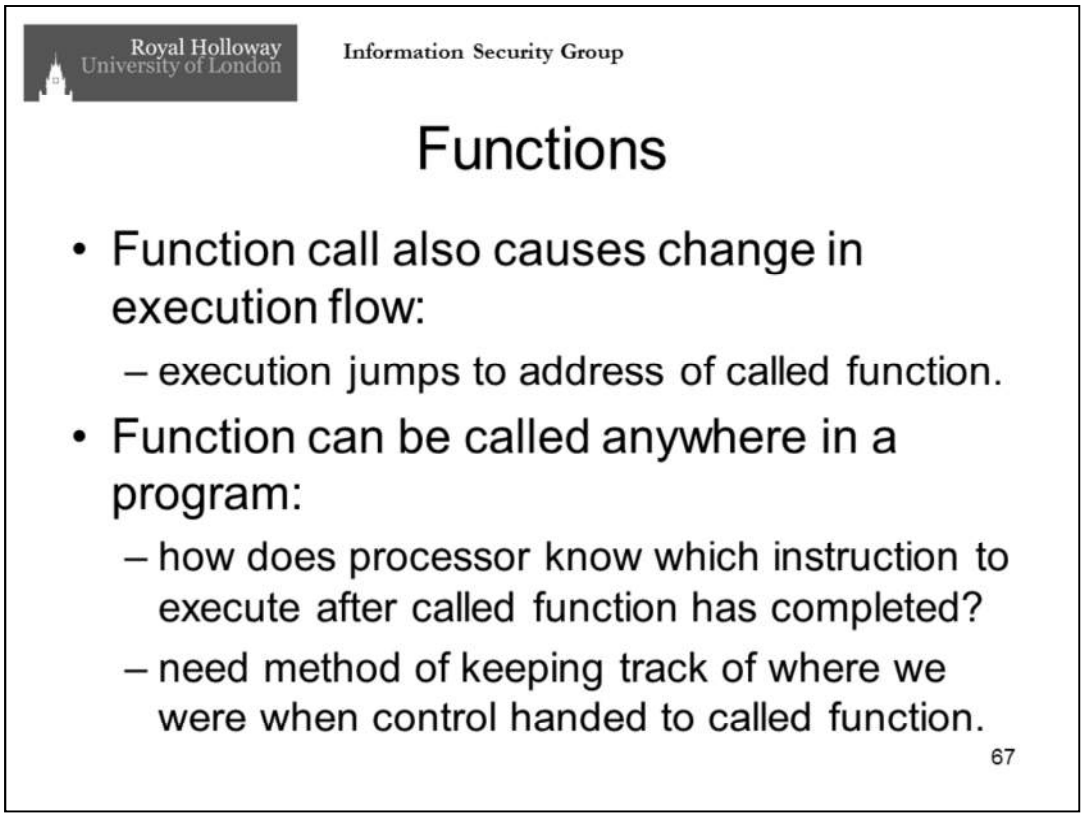
- the Sun Microsystems XDR library;
- the Windows DirectX library quartz.dll;
- bash (open source Unix shell).

These vulnerabilities often arise because of an incorrect implementation of input validation.



There are two main ways of controlling the flow (the execution path) of a program:

- **selection**:
  - choose between two different execution paths based on the result of a logical test, e.g.:
    - `if ... then ... else ...`
- **iteration**:
  - repeat a sequence of instructions all the while a logical test evaluates to true, e.g.:
    - `while ... do ...`
    - `for` loops
    - `repeat ... until ...`



A function call also causes a change in the flow of execution:

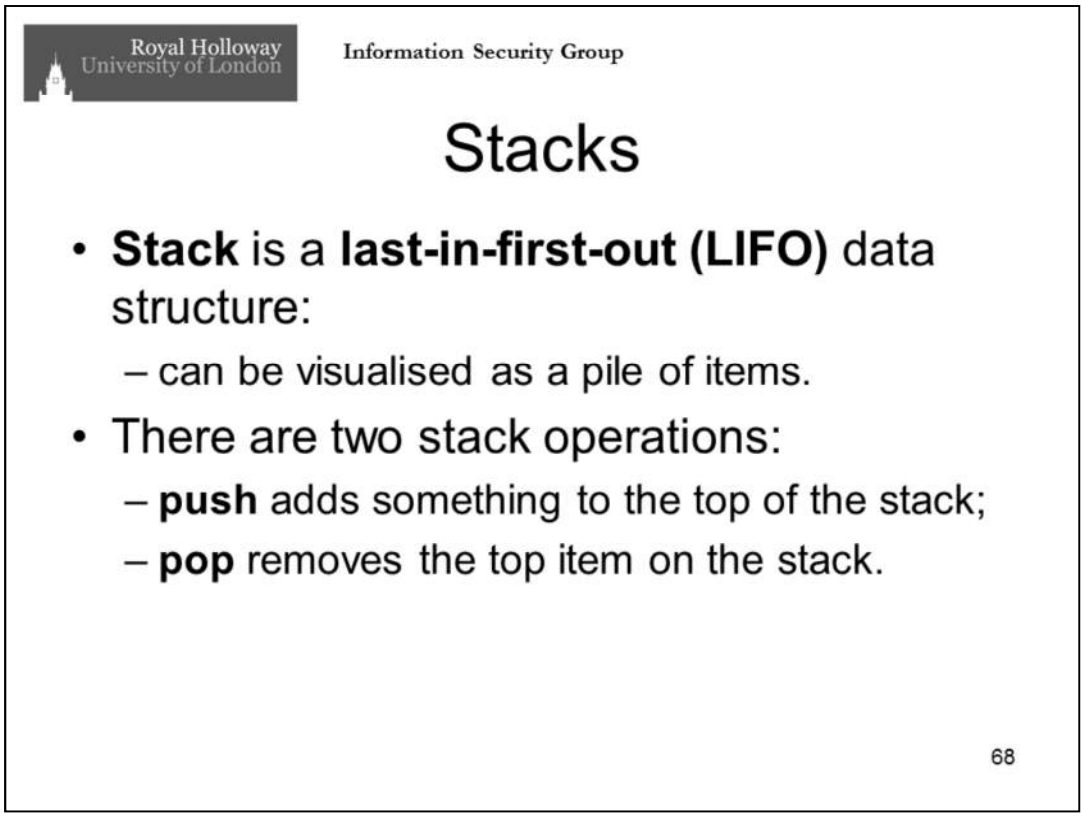
• execution jumps to the first address of the called function.

A function can be called at any point in a program:

• how does the processor know which instruction to execute after the called function has completed?

• we need a method for keeping track of where we were in the calling function when it handed control to the called function.

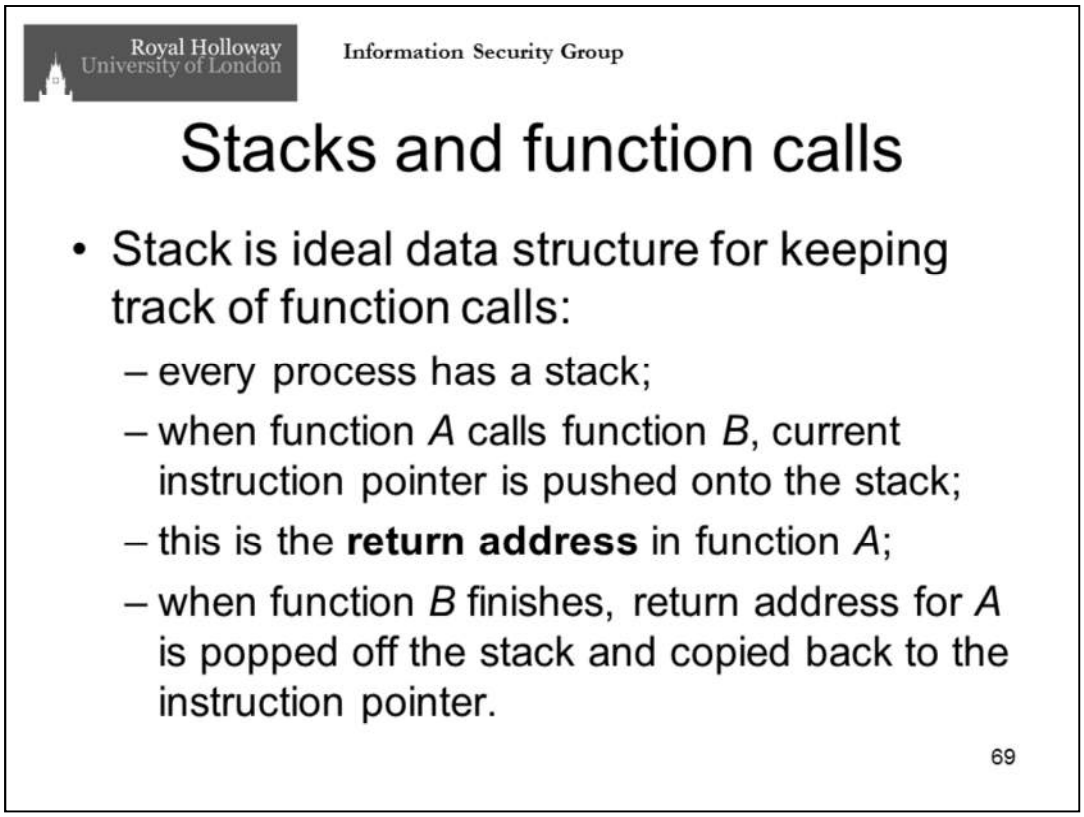
The standard solution to this 'housekeeping problem' is the use of a data structure called a **stack**.



A **stack** is a **last-in-first-out (LIFO)** data structure, i.e. a way of organising data in a computer. It operates analogously to a pile of items.

There are two fundamental operations that can be performed on a stack, namely:

- **push** adds something to the top of the stack;
- **pop** removes the top item on the stack.



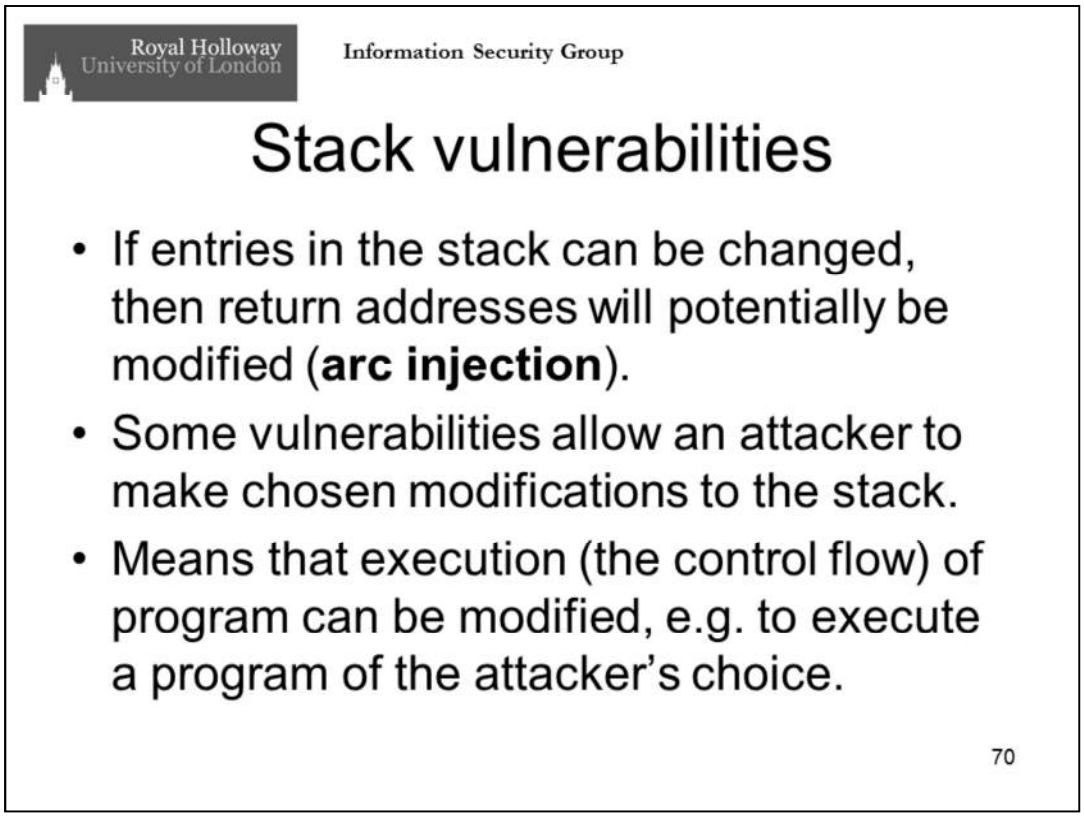
A stack is an ideal data structure for keeping track of function calls. As a result every process address space includes a stack region;

• when a function *A* calls a function *B*, the contents of the instruction pointer (the address of a location in computer memory) is pushed onto the stack;

• this address is the **return address** in function *A*;

• when function *B* finishes the return address for *A* is popped off the stack and copied back to the instruction pointer.

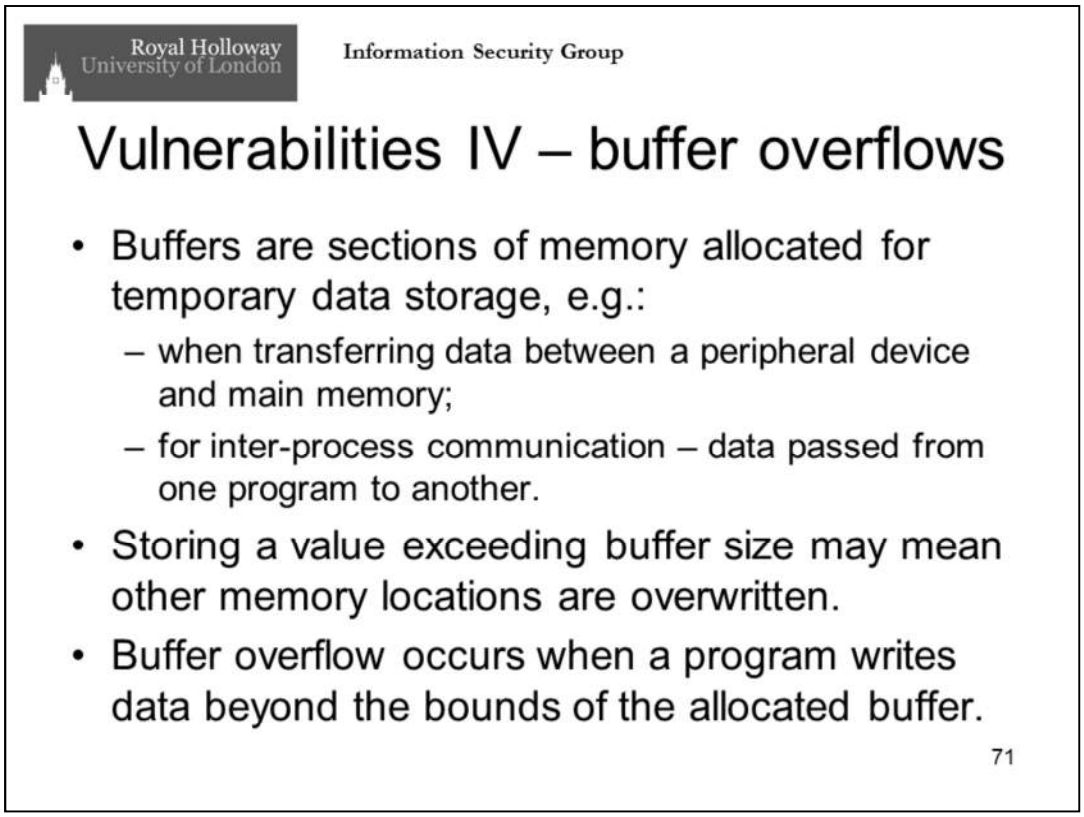
The 'last in first out' property of the stack allows for the simple management of an indefinite number of function calls.



If entries in the stack can be changed, then return addresses will potentially be modified. This is what is known as an **arc injection**.

Some vulnerabilities allow an attacker to make chosen modifications to the stack.

Changing the stack would mean that the execution (the control flow) of the program would be modified, e.g. causing the execution of a program of the attacker's choice.



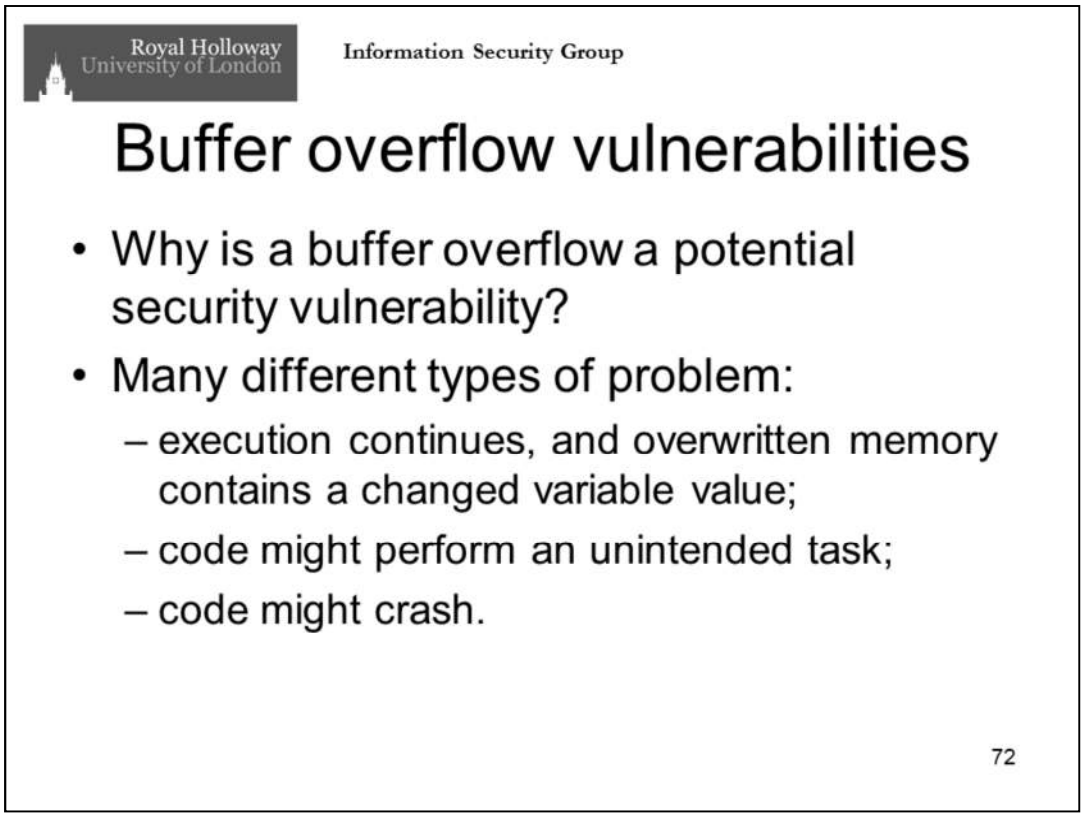
The allocation of memory resources is probably the most common area in which software security vulnerabilities arise. The majority of these problems arise from **buffer overflows**.

Buffers are sections of memory allocated for data storage, typically on a temporary basis while data is transferred from one location to another, e.g.:

• when transferring data from a peripheral device to main memory and vice versa;

• for inter-process communication, allowing parameters to pass from one program to another.

Storage of a value exceeding the size of the buffer may result in other memory locations being overwritten. A buffer overflow (or buffer over-run) occurs when a program writes data beyond the bounds of the allocated buffer.

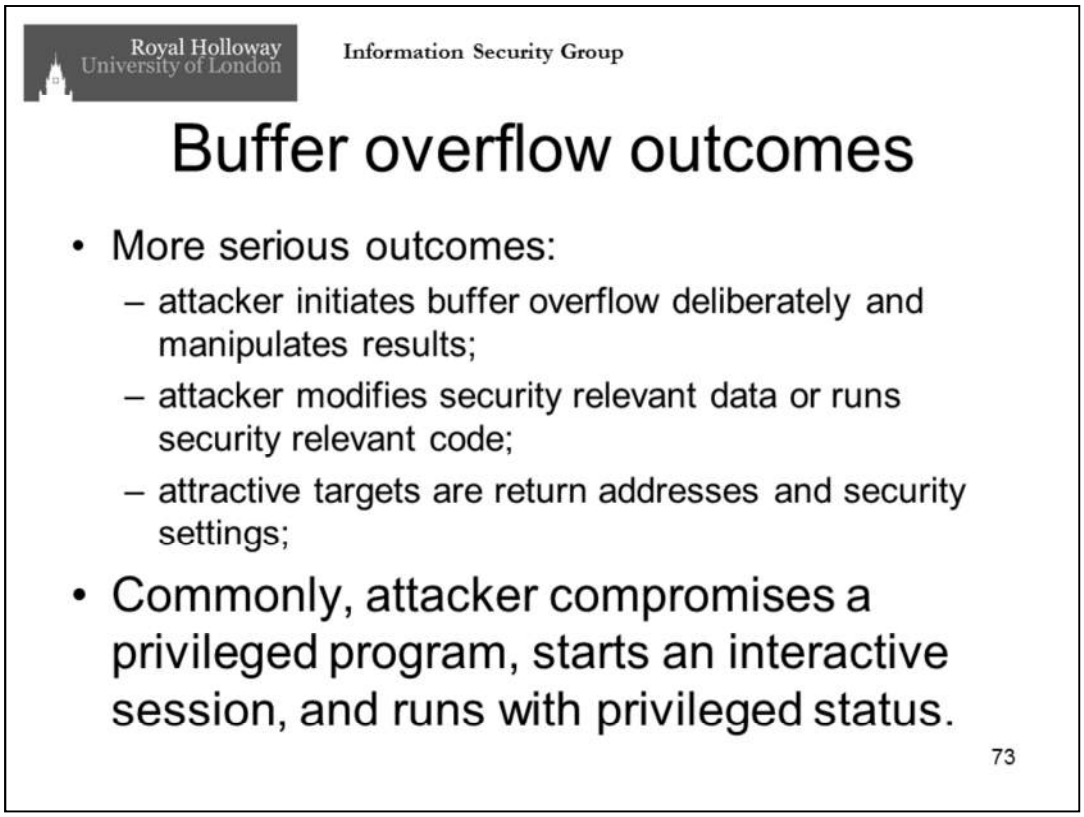


Why is a buffer overflow a potential security vulnerability? Many different types of problem can result from a buffer overflow, including:

• execution could continue, and the overwritten memory will contain a changed variable value;

• the code might perform an unintended task;

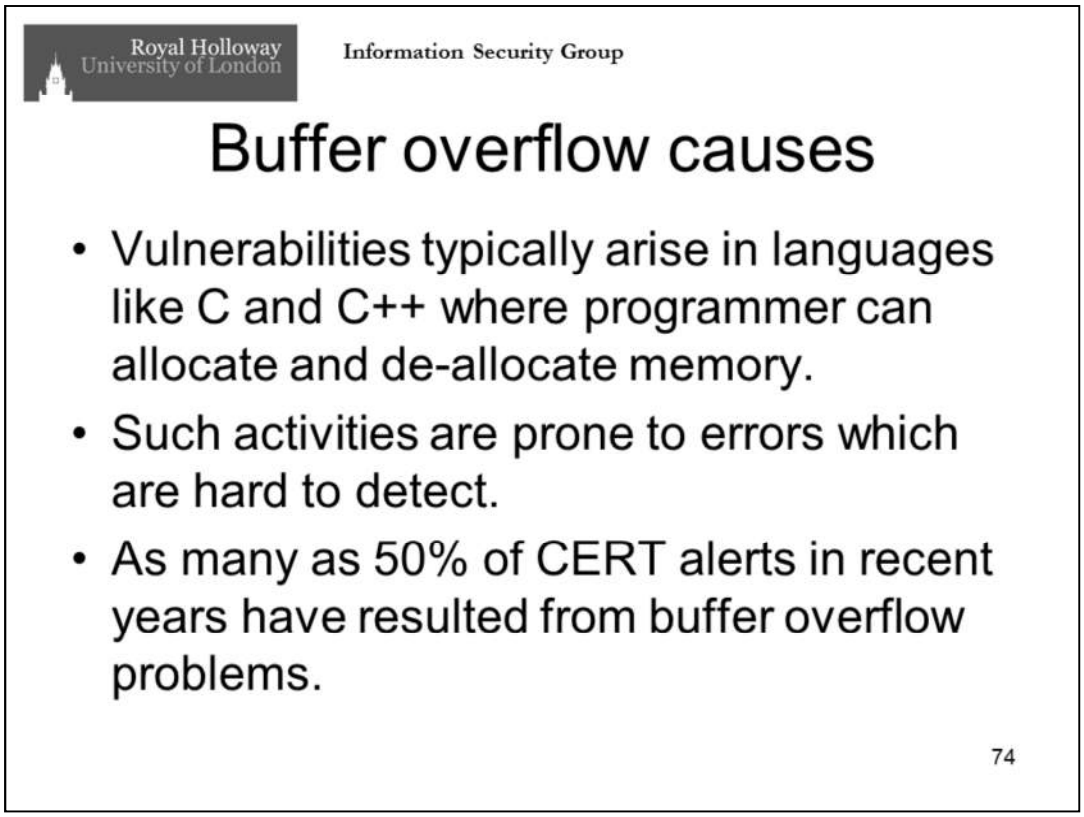
• the code might crash.



More serious possible outcomes include the following:

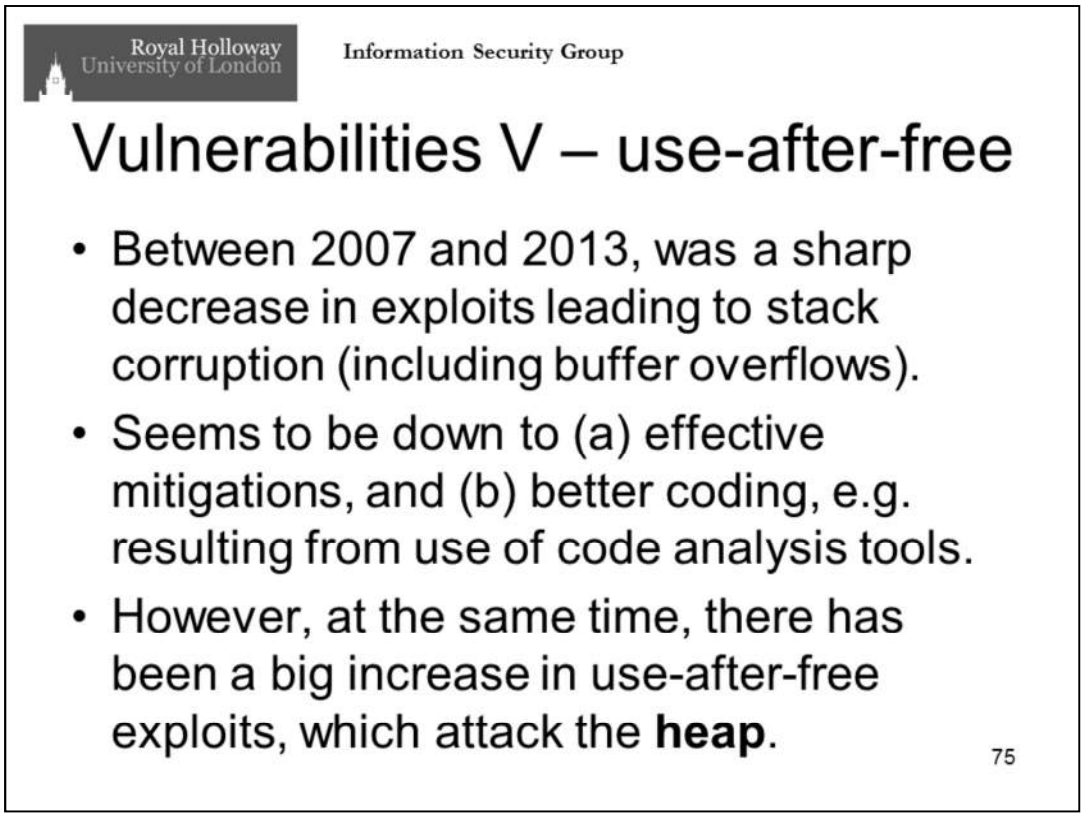
- an attacker might initiate the buffer overflow deliberately and manipulate the results for his/her own ends;
- an attacker might modify security relevant data or run security relevant code;
- attractive targets for such attacks (i.e. types of data to be over-written) include return addresses and security settings.

Commonly an attacker compromises a privileged program, establishes an interactive session and runs with privileged status. With enhanced privileges the attacker can have complete control of the machine.



Such vulnerabilities typically arise with the use of programming languages like C and C++, where the programmer is free to allocate and de-allocate memory. Such activities are prone to errors which are hard to detect.

As many as 50% of CERT alerts in recent years have resulted from buffer overflow problems.

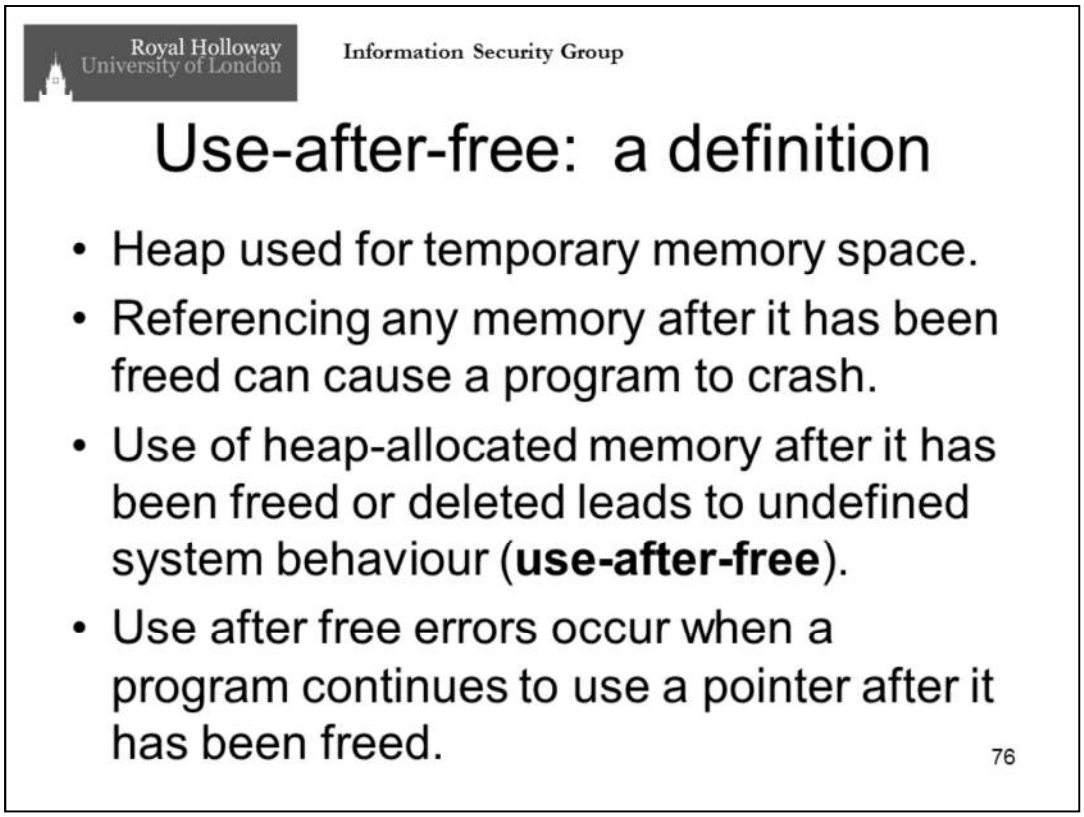


As reported in Volume 16 of Microsoft's Security Intelligence Report (SIR), in the years up to 2013 a sharp decrease was seen in exploits leading to stack corruption, including stack-based buffer overflows. This class of vulnerabilities accounted for 54% of known exploited Microsoft remote code execution CVEs (Common Vulnerabilities and Exposures) in 2007, but just 5% in 2013. This change seems to be due to a combination of mitigations and better coding, including the use of code analysis tools.

At the same time, an increasing number of exploits of use-after-free vulnerabilities have been exploited. These exploits focus on the **heap** instead of the stack.

See: http://www.microsoft.com/security/sir/default.aspx for the SIR. Volume 16 is an extremely interesting read.

Note that the reference here to CVE is to those vulnerabilities that are published in the US Government's National Vulnerability Database (NVD) – see nvd.nist.gov. All vulnerabilities in the NVD are given a CVE identifier (e.g. CVE2012-1889).



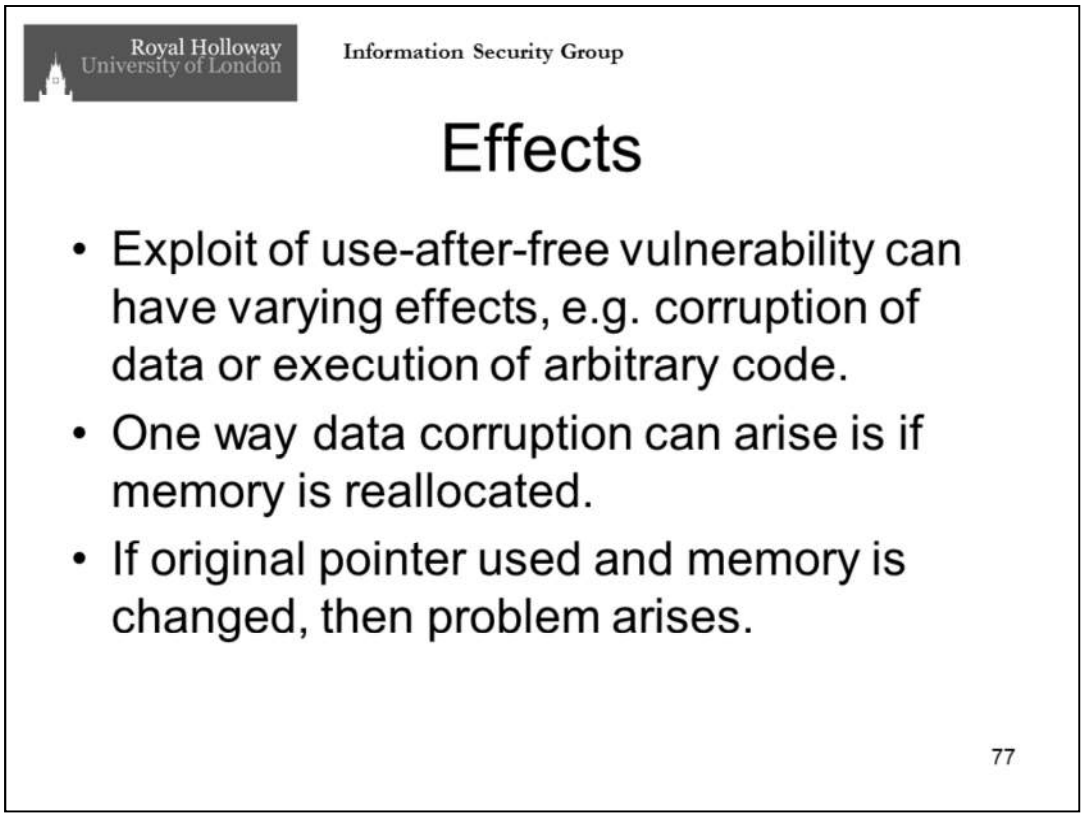
The **heap** for a process consists of memory space that is allocated for temporary use.

Referencing memory after it has been freed (i.e. after it is no longer required for the purpose for which it was allocated) can cause a program to crash. The use of heap-allocated memory after it has been freed or deleted leads to undefined system behaviour (this is what is meant by **use-after-free**).

Use-after-free errors occur when a program continues to use a pointer after it has been freed. Use-after-free errors have two common and sometimes overlapping causes:

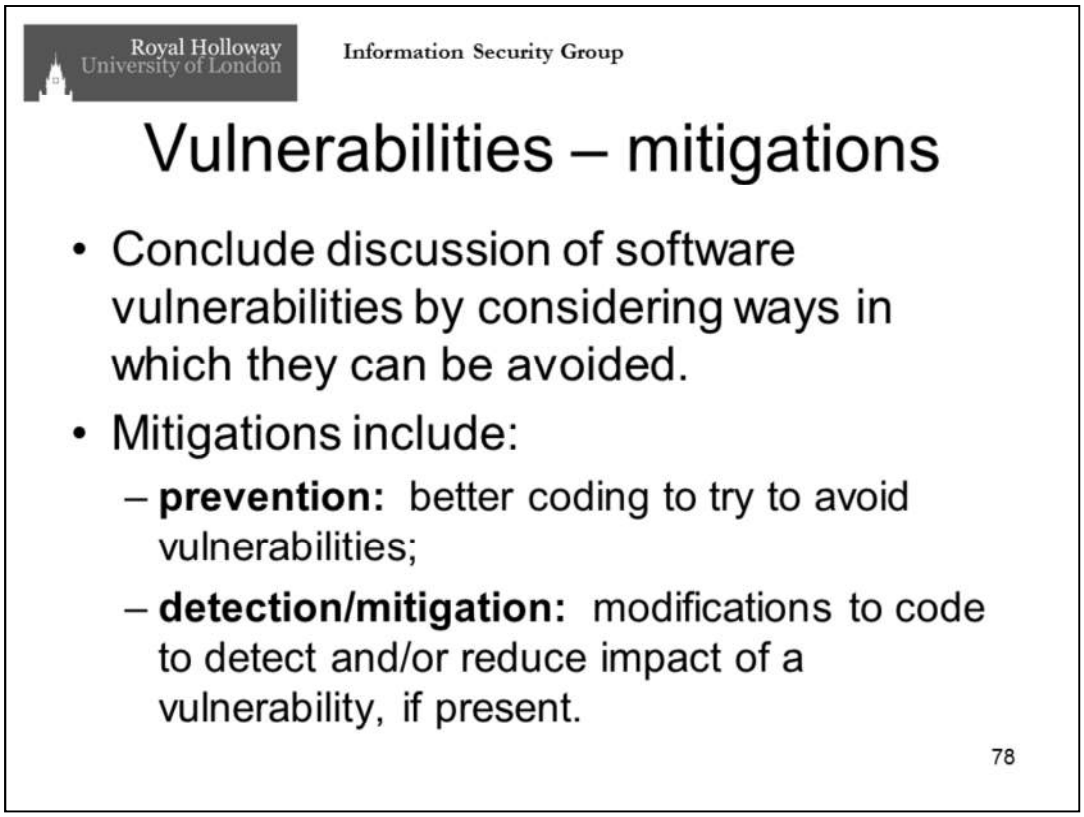
- error conditions and other exceptional circumstances;
- confusion over which part of the program is responsible for freeing the memory.

[Thanks to www.owasp.org for their explanation of use-after-free, on which this and the next slide have been based. For further information see also the wikipedia article on dangling pointers.]



The use of previously freed memory can have any number of adverse consequences – ranging from the corruption of valid data to the execution of arbitrary code, depending on the instance and time of occurrence of the flaw.

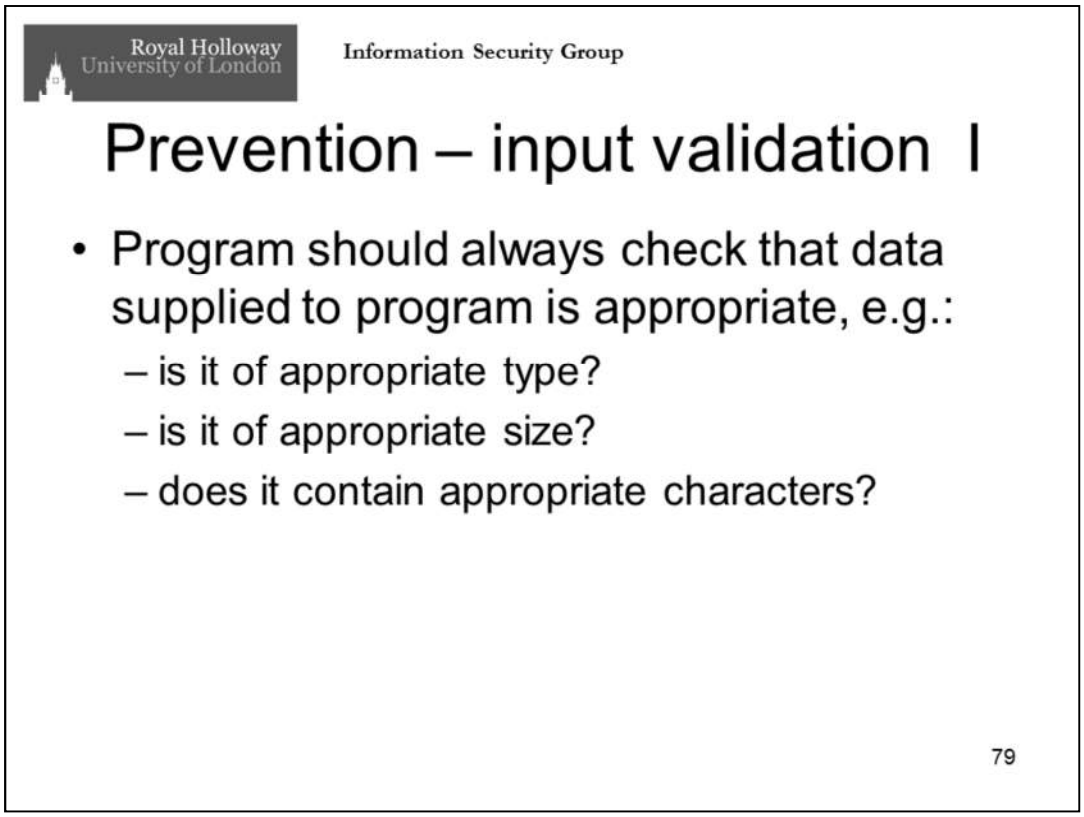
One of the simplest ways in which data corruption can occur involves the system's reuse of the freed memory. In this scenario, the memory in question is allocated to another pointer validly at some point after it has been freed. The original pointer to the freed memory is used again and points to somewhere within the new allocation. If use of the original pointer causes the data to be changed, this will corrupt the validly used memory; this will induce undefined behaviour in the process.



We conclude this discussion of software vulnerabilities by considering ways in which they can be avoided. Possible mitigations include:

• **prevention:** better coding techniques can be used to help avoid the vulnerabilities being present;

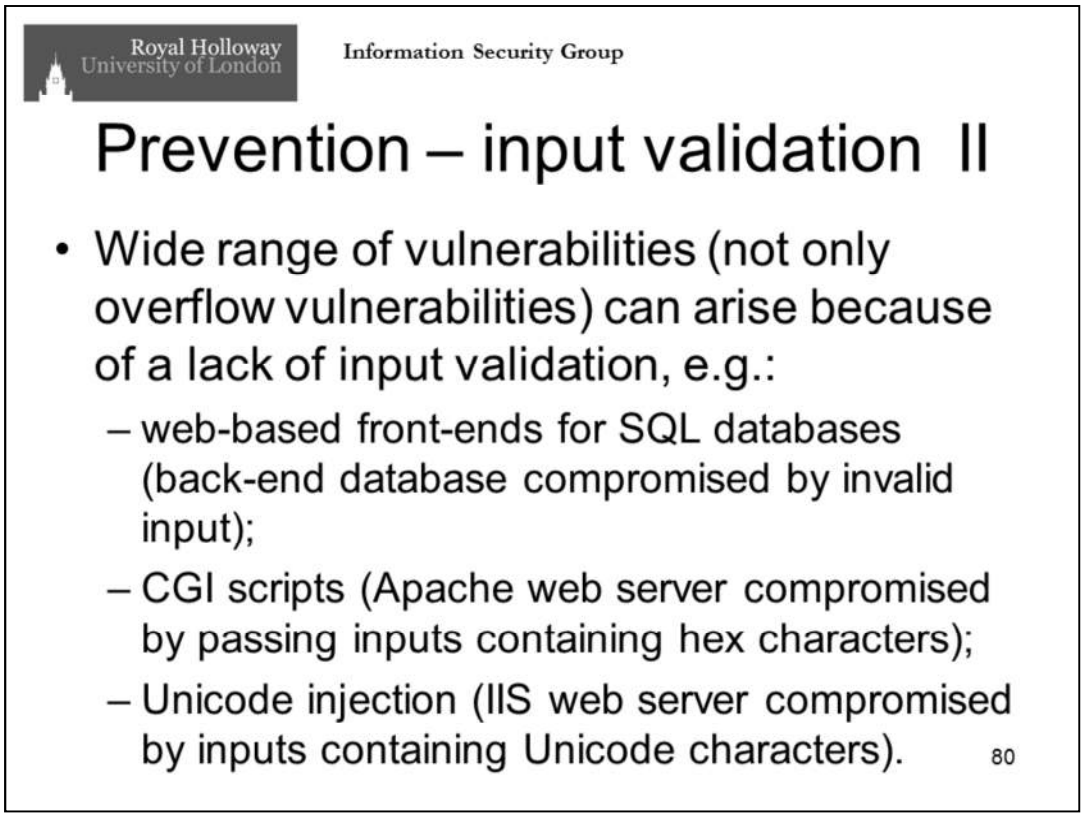
• **detection/mitigation:** modifications can be made to the code to detect and/or to try to reduce the impact of a vulnerability if it is present.



A program should always check that any external data supplied to the program is appropriate, e.g.:

- is it of an appropriate type?
- is it of an appropriate size?
- does it contain appropriate characters?

Such validation checks are in many cases simple to perform, but are very often neglected.

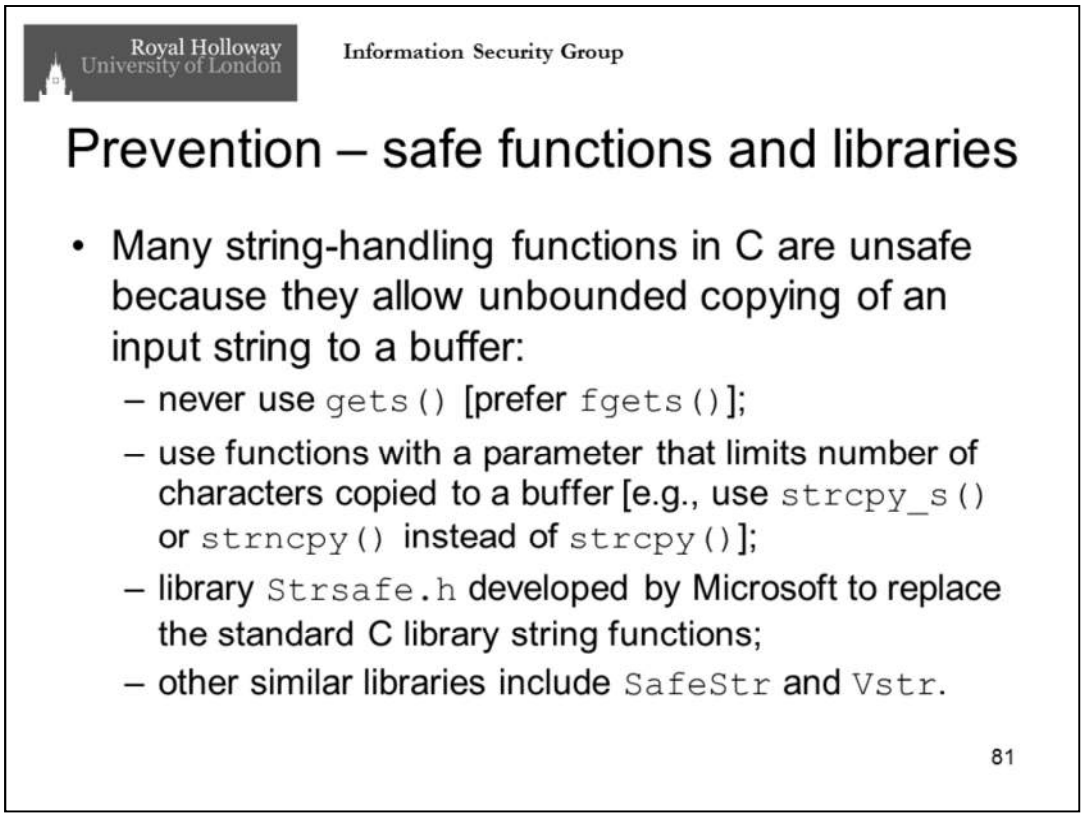


A wide range of vulnerabilities (not only overflow vulnerabilities) can arise because of a lack of input validation. For example, vulnerabilities of this type can arise in:

• web-based front-ends for SQL databases (where the back-end database is compromised by input inadequately validated by front end) – see, for example:
http://cert.uni-stuttgart.de/advisories/apache_auth.php
and
http://www.us-cert.gov/cas/techalerts/TA04-160A.html

• CGI scripts (e.g. where an Apache web server was compromised by passing inputs containing hexadecimal characters) – see:
http://www.cert.org/advisories/CA-1996-06.html

• Unicode injection (e.g. where an IIS web server was compromised by passing inputs containing Unicode characters) – see:
http://www.kb.cert.org/vuls/id/111677



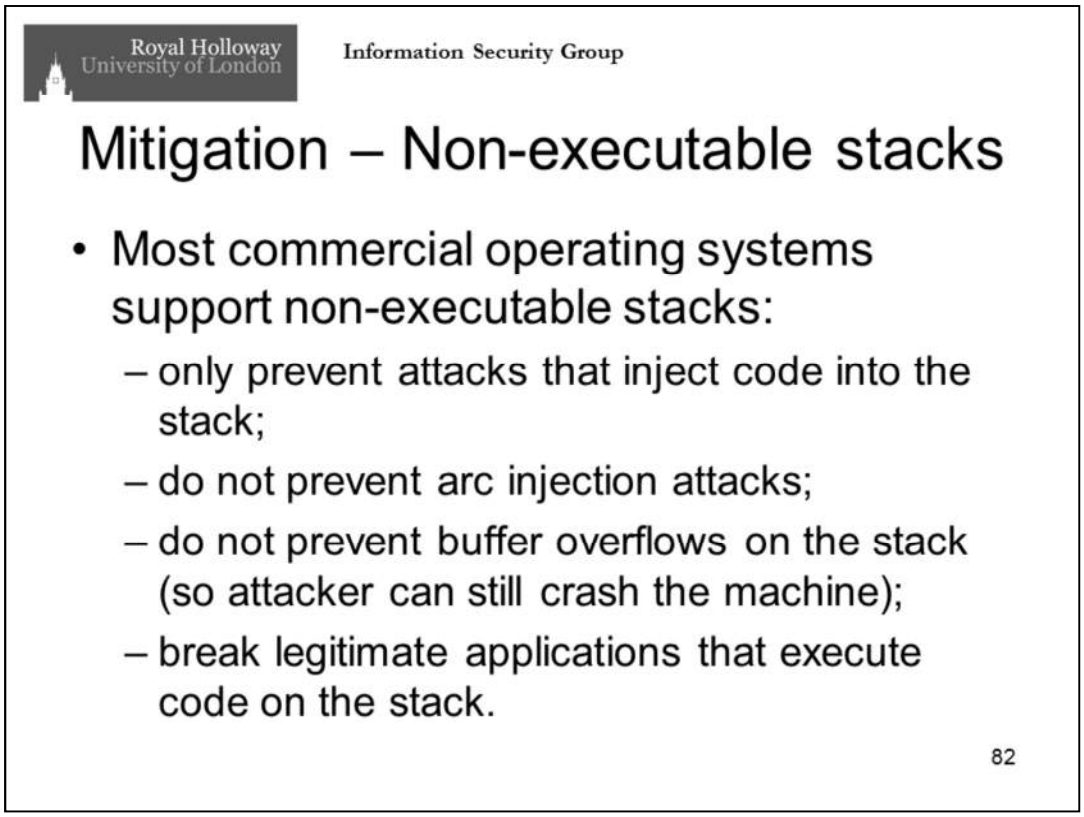
Many string-handling functions in C are unsafe because they allow unbounded copying of an input string to a buffer:

• never use `gets()` (use `fgets()` instead);

• use functions that take a parameter that limits the number of characters that are copied to a buffer (for example, use `strcpy_s()` or `strncpy()` instead of `strcpy()`);

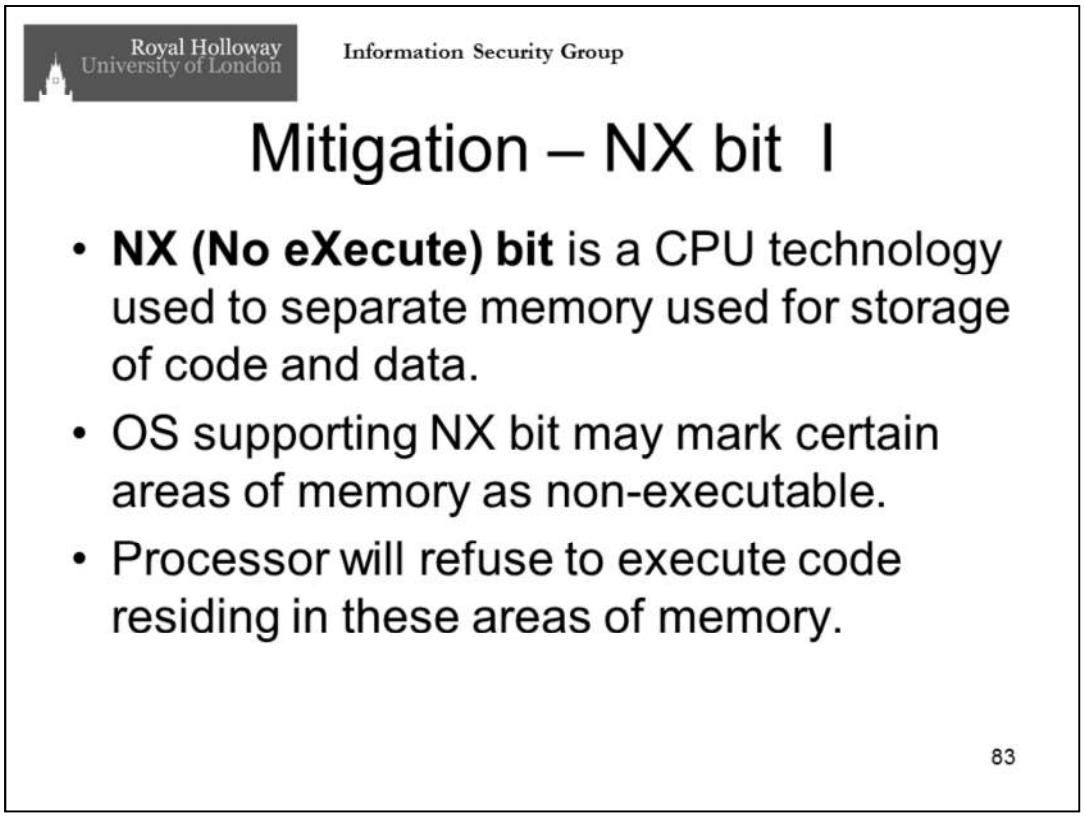
• the library `Strsafe.h` has been developed by Microsoft to replace the standard C library string functions;

• other similar libraries of 'safe' string-handling functions include `SafeStr` and `Vstr`.



Most commercial operating systems support non-executable stacks, i.e. where the processor will refuse to execute if the program counter points to a stack location. Such precautions:

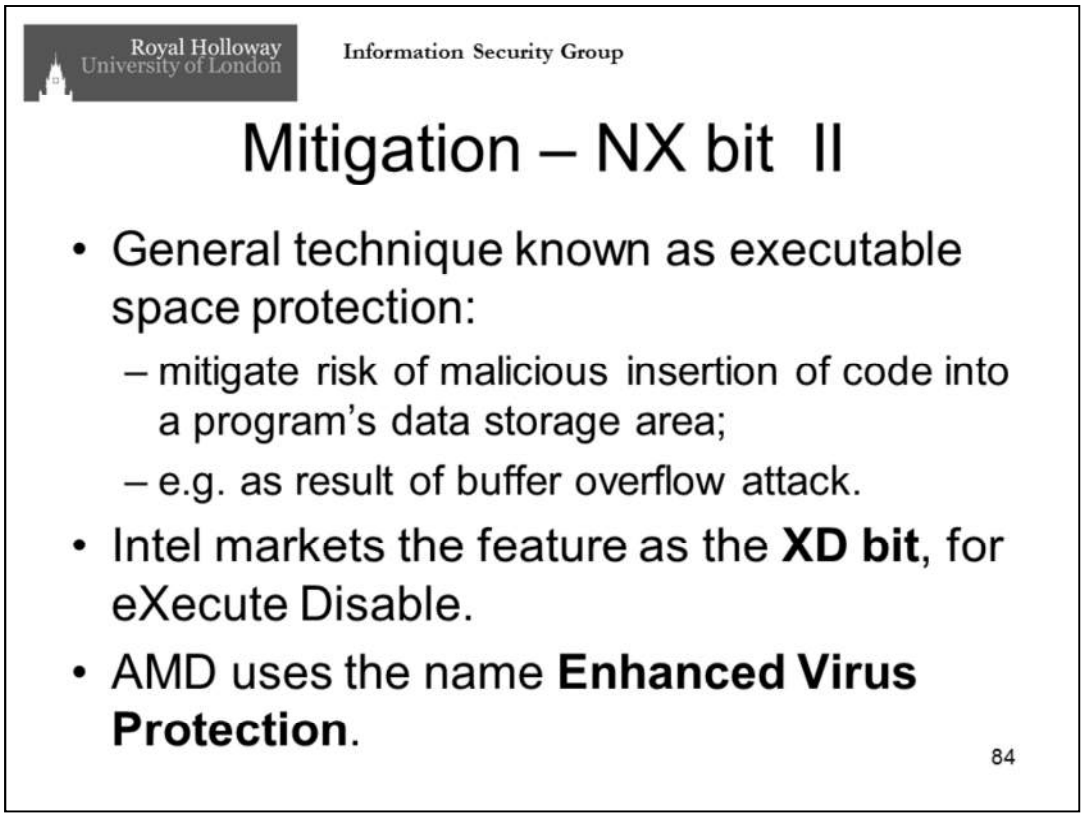
- only prevent attacks that inject code into the stack;
- do not prevent arc injection attacks (i.e. where return addresses are modified);
- do not prevent buffer overflows on the stack (so an attacker can still crash the machine);
- break legitimate applications that execute code on the stack.



The **NX (No eXecute) bit** is a technology used in CPUs to segregate areas of memory used for storage of processor instructions (code) and for data.

An OS supporting the NX bit may mark certain areas of memory as non-executable.

The processor will then refuse to execute code residing in these areas of memory.



The general technique is known as executable space protection. It is used to prevent execution of code maliciously inserted into a program's data storage area, e.g. as a result of a buffer overflow attack.

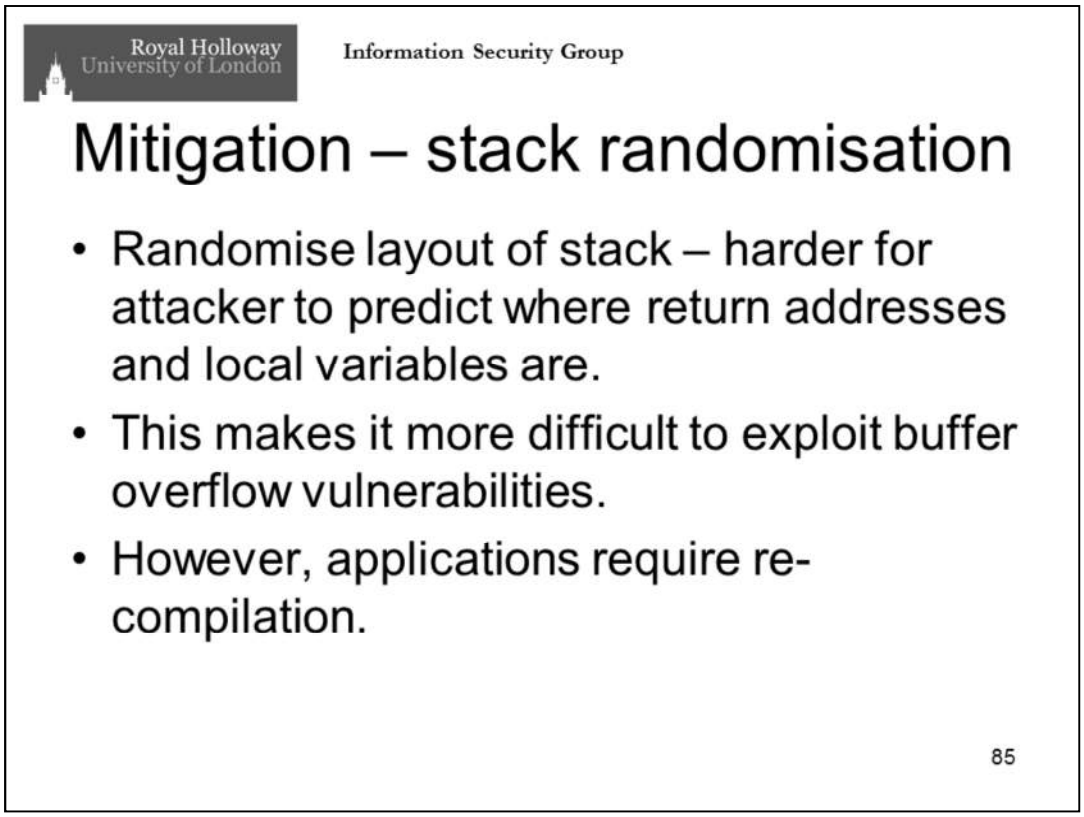
Intel markets the feature as the **XD bit**, for eXecute Disable. AMD uses the name **Enhanced Virus Protection**.

Microsoft Windows uses NX protection. In Windows XP and Server 2003 (and later versions of Windows), the feature is called Data Execution Prevention (DEP). If the processor supports this feature in hardware, then the NX features are turned on in Windows XP/Server 2003 by default.

However, even if the processor does not support it, DEP has a 'software-based mode', which attempts to emulate the non-executable stack protection even without support from the processor. Software-enabled DEP is weaker than the hardware-enforced version, but still provides a level of protection.

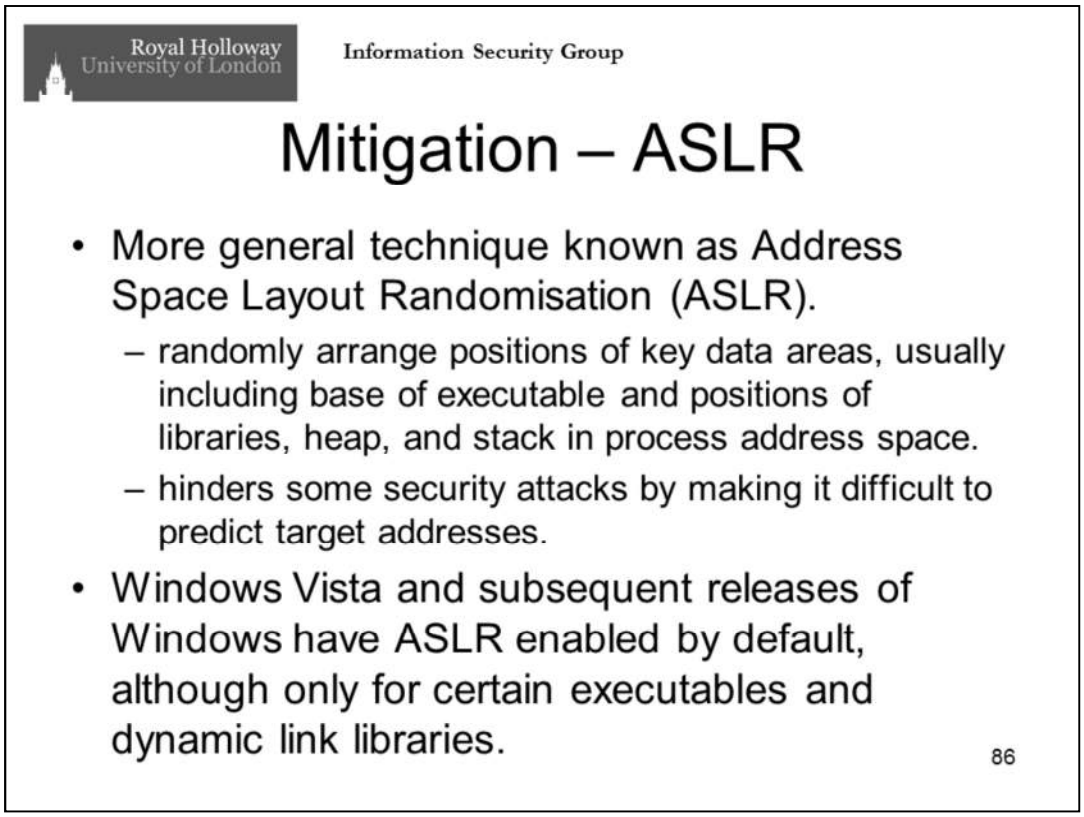
For further information about DEP see:

http://windows.microsoft.com/en-US/windows-vista/Data-Execution-Prevention-frequently-asked-questions



Randomising the layout of the stack makes it harder for the attacker to predict where return addresses and local variables will be. This makes it more difficult to exploit buffer overflow vulnerabilities.

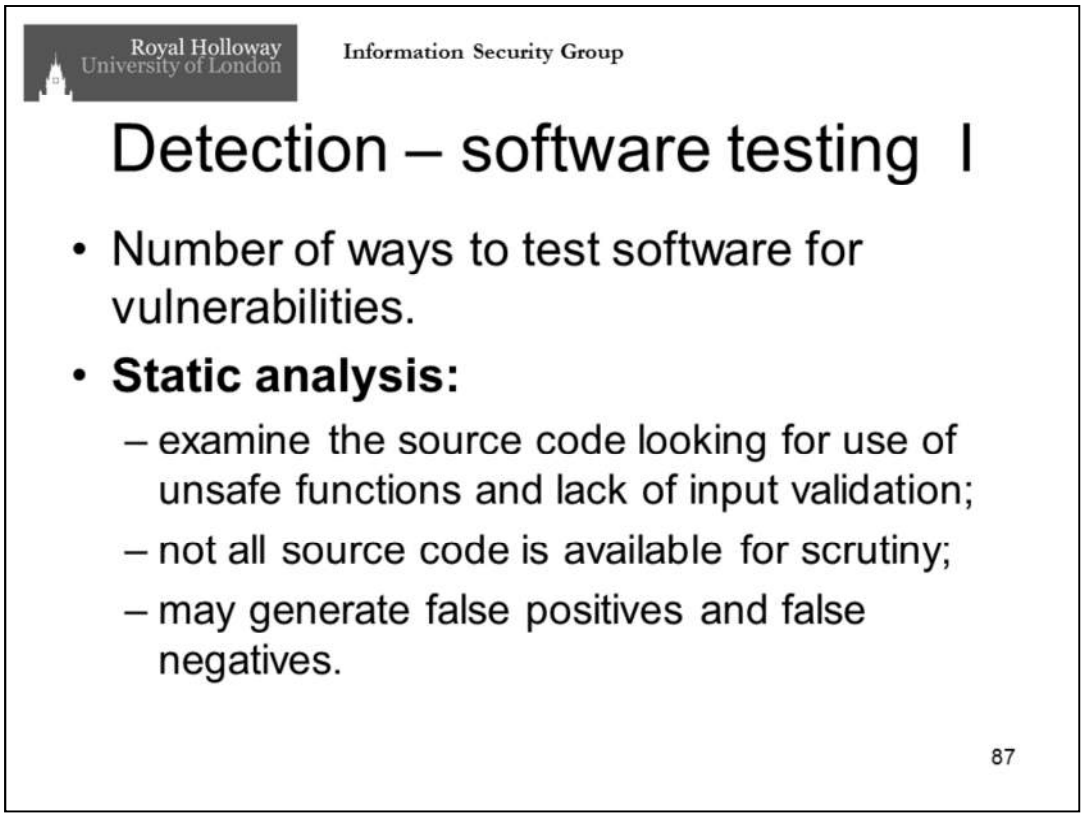
However, for this to be effective applications require re-compilation.



A more general technique of this type is known as Address Space Layout Randomisation (ASLR). It involves randomly arranging the positions of certain key data areas, usually including the base of the executable and the positions of libraries, heap, and stack in the process's address space.

It hinders some security attacks by making it difficult to predict target addresses.

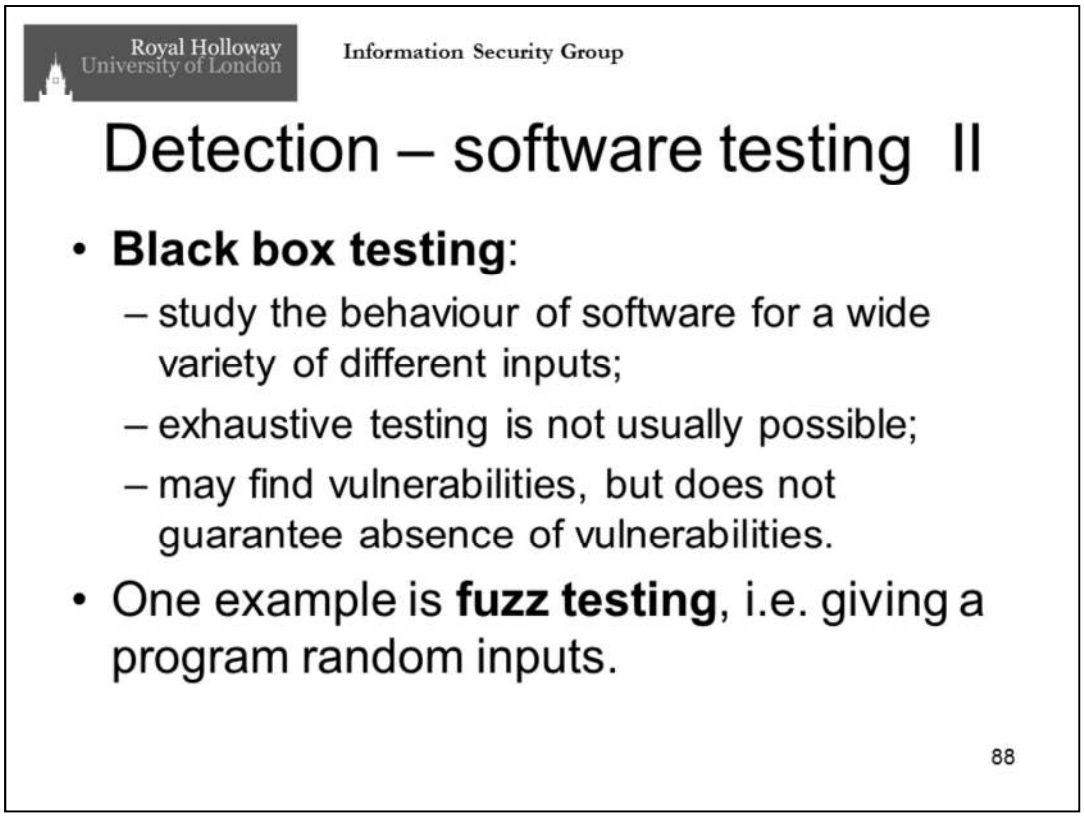
Windows Vista and subsequent releases of Windows have ASLR enabled by default, although only for certain executables and dynamic link libraries.



There are a number of ways of testing software for vulnerabilities. We briefly mention two such possibilities.

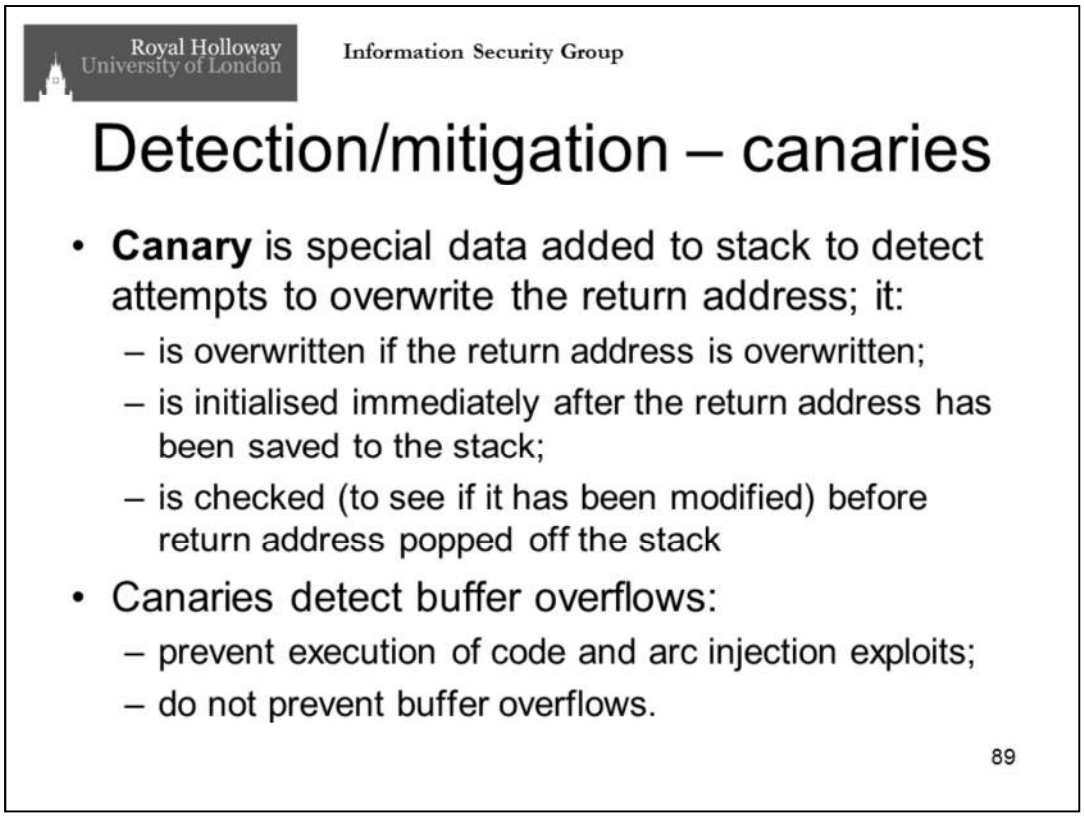
Static analysis involves examining the source code to look for the use of unsafe functions and lack of input validation.

However not all source code is available for scrutiny. Such an approach may also generate false positives and false negatives.



Black box testing involves studying the behaviour of software for a wide variety of different inputs. Exhaustive testing (of all possible inputs) is not usually possible. Such an approach may find vulnerabilities, but does not guarantee the absence of vulnerabilities.

One particular example of black box testing is known as **fuzz testing**, where a program is executed many times with randomly selected inputs (possibly semi-structured).



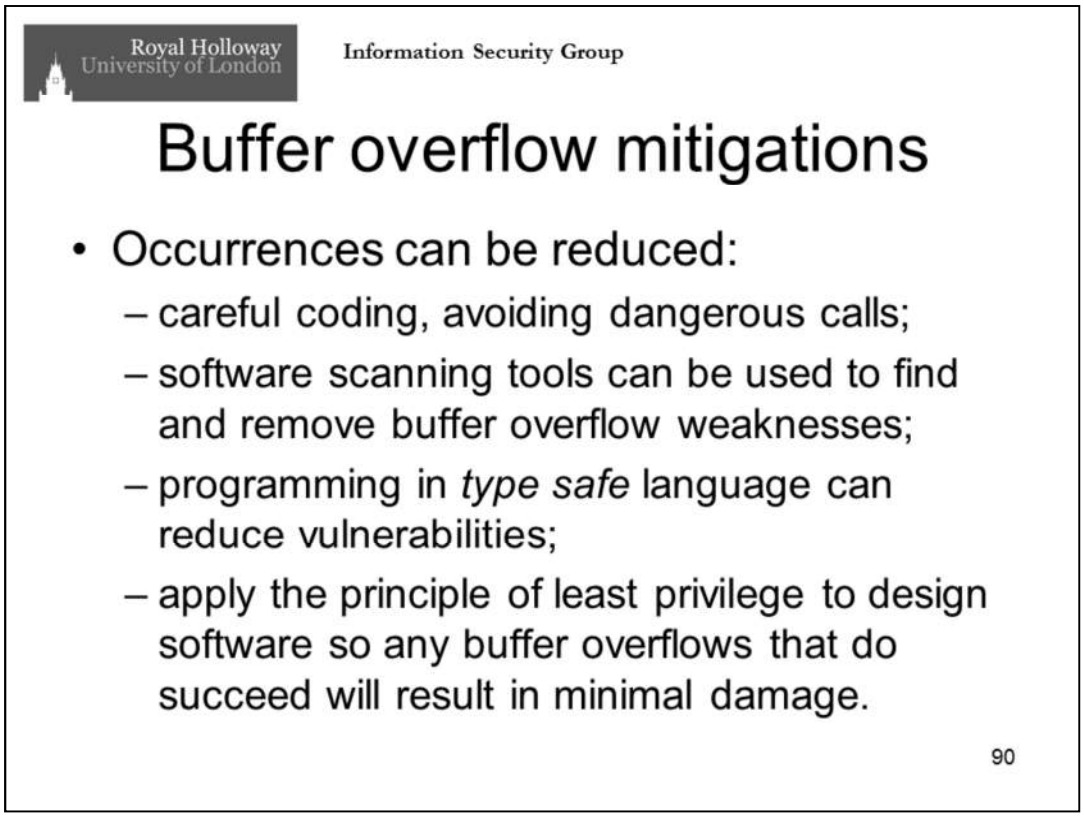
A **canary** is a special piece of data added to the stack to detect attempts to overwrite the return address. It:

- will be overwritten if the return address is overwritten;
- is initialised immediately after the return address has been saved to the stack;
- is checked (to see if it has been modified) before the return address popped off the stack

Canaries can be used to detect buffer overflows:

- can help prevent execution of code and arc injection exploits.

However, they do not prevent buffer overflows.



Occurrences of buffer overflows can be reduced:

- by careful coding, avoiding dangerous calls;
- software scanning tools can be used to find and remove buffer overflow weaknesses;
- programming in a **type safe** language can reduce vulnerabilities (see later slides);
- apply the principle of least privilege to design software so any buffer overflows that do succeed will result in minimal damage.

Other tools and techniques are available to programmers. Users of software susceptible to buffer overflow attacks should be aware of the security vulnerability alerts from organisations such as CERT, and follow any relevant advice.

A helpful review of buffer overflow attacks and mitigations can be found in:

B. M. Padmanabhuni and H. B. K. Tan, Defending against buffer-overflow vulnerabilities. *IEEE Computer*, **44 no. 11** (November 2011), pp.53-60.

[Download a copy at:

http://www.chrismitchell.net/IY5512/mco2011110053.pdf].

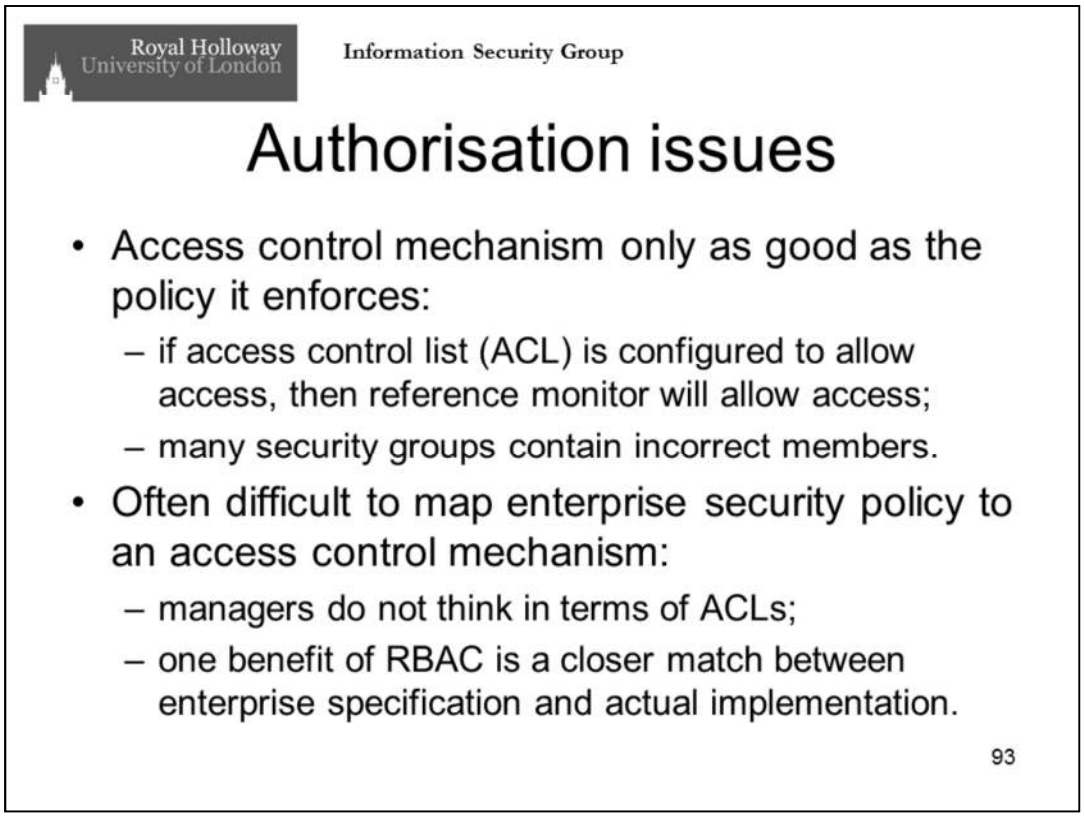
Inadequacies in programming are not the only sources of vulnerabilities in software.

One significant source of vulnerabilities is password-based authentication:

- users often make poor choices for passwords;
- users often inadequately protect their passwords.

Implementations of authentication services are sometimes flawed:

- in the past, Unix and Windows have both used authentication methods with small key spaces;
- widely used protocols (e.g. Kerberos) are vulnerable to off-line dictionary attacks, when passwords are poorly chosen.



An access control mechanism is only as good as the policy it enforces:

• if an access control list (ACL) is configured to allow access, then the reference monitor will allow access;

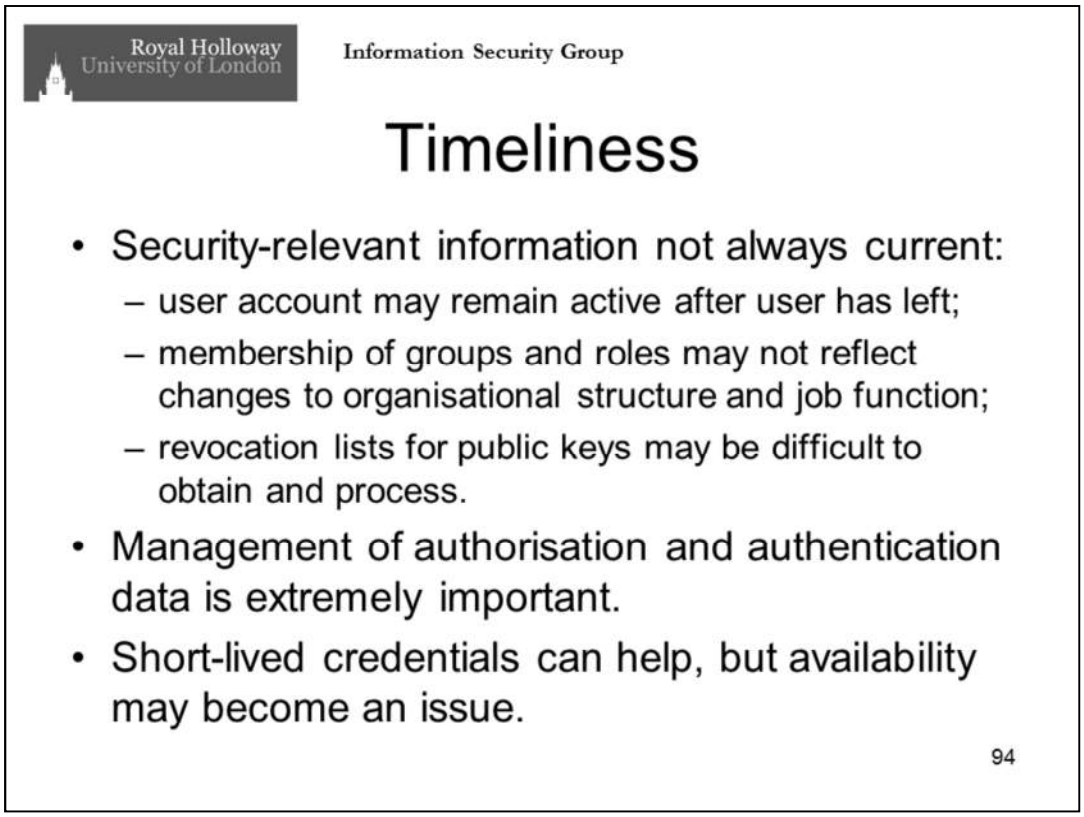
• in practice, access control systems are often misconfigured; for example, security groups may contain the wrong members.

It is often difficult to map an enterprise security policy to the specification scheme provided by the access control mechanism:

• system managers do not think in terms of ACLs;

• one benefit of role based access control (RBAC) is a closer match between an enterprise specification and the actual implementation.

[Note that we examine ACLs and RBAC later in the course].

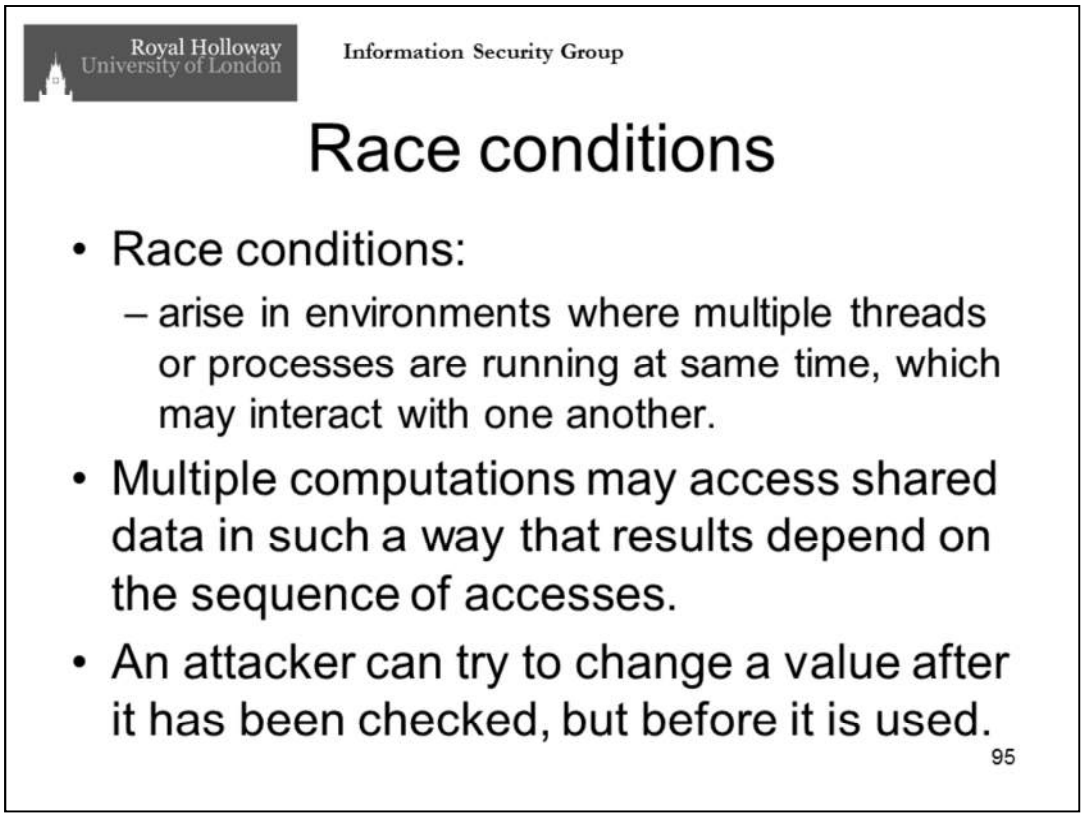


Security-relevant information may not always be up to date:

- a user account may remain active long after the user has left organisation;
- membership of groups and roles may not reflect changes to organisational structure and job function;
- revocation lists for public keys may be difficult to obtain and process.

These vulnerabilities mean that management of authorisation and authentication data is extremely important.

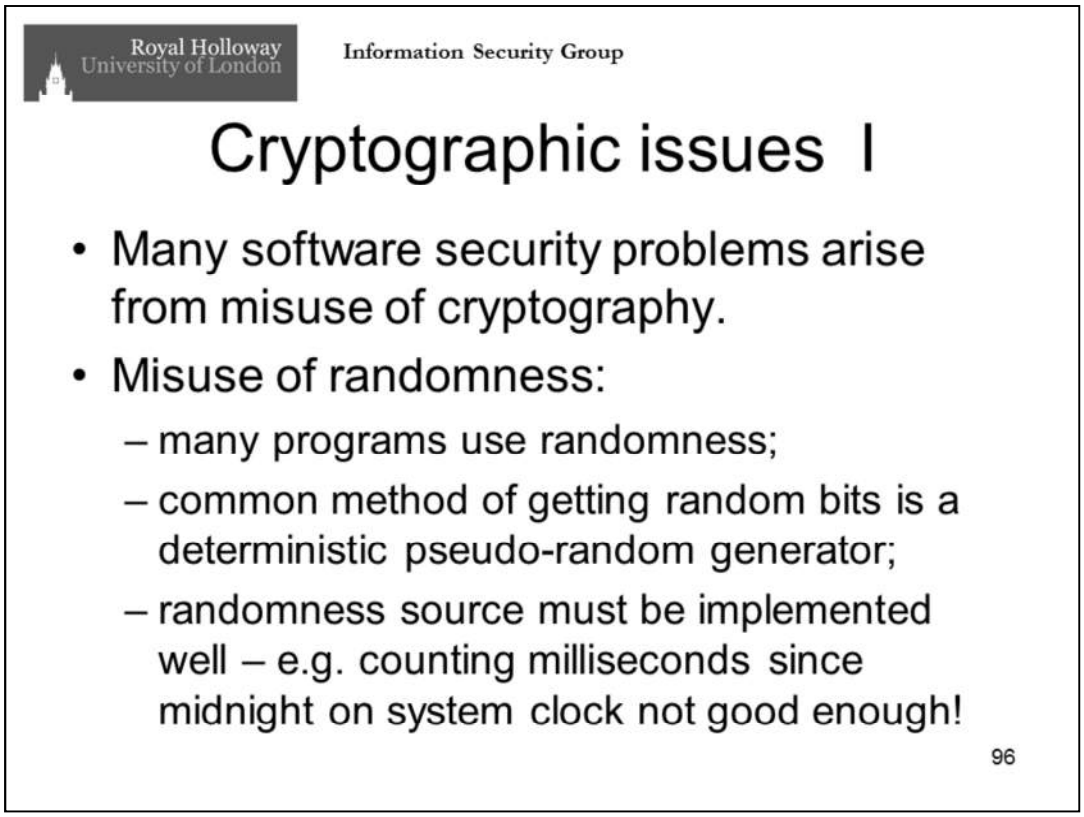
Short-lived credentials can help, but availability may become an issue if credentials are allowed to expire without replacement.



A **race condition** is a flaw in a process where the result of the process is dependent on the sequence or timing of other events. The term originates with the idea of two signals in a circuit 'racing each other' to influence the output.

Race conditions in computer systems arise in environments where multiple threads or processes are running at the same time, which may interact with one another. Multiple computations may access shared data in such a way that their results depend on the sequence of accesses.

An attacker can try to change a value after it has been checked, but before it is used.



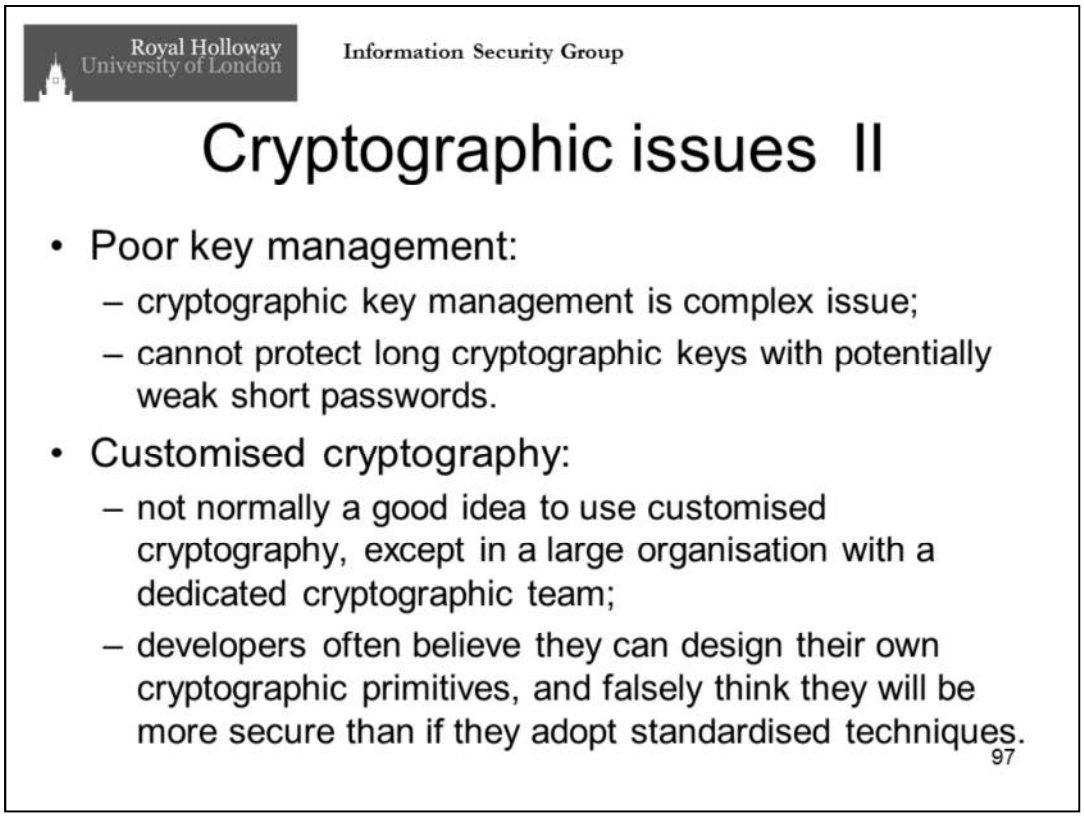
Many software security problems arise from misuse of cryptography.

Misuse of randomness is common:

• many programs require sources of randomness (e.g. to generate keys for cryptographic algorithms, or random challenges in protocols);

• the most common method of generating 'randomness' is to use a deterministic pseudo-random generator;

• a randomness source must be designed and implemented well – simply counting the number of milliseconds since midnight on the system clock is not normally good enough!

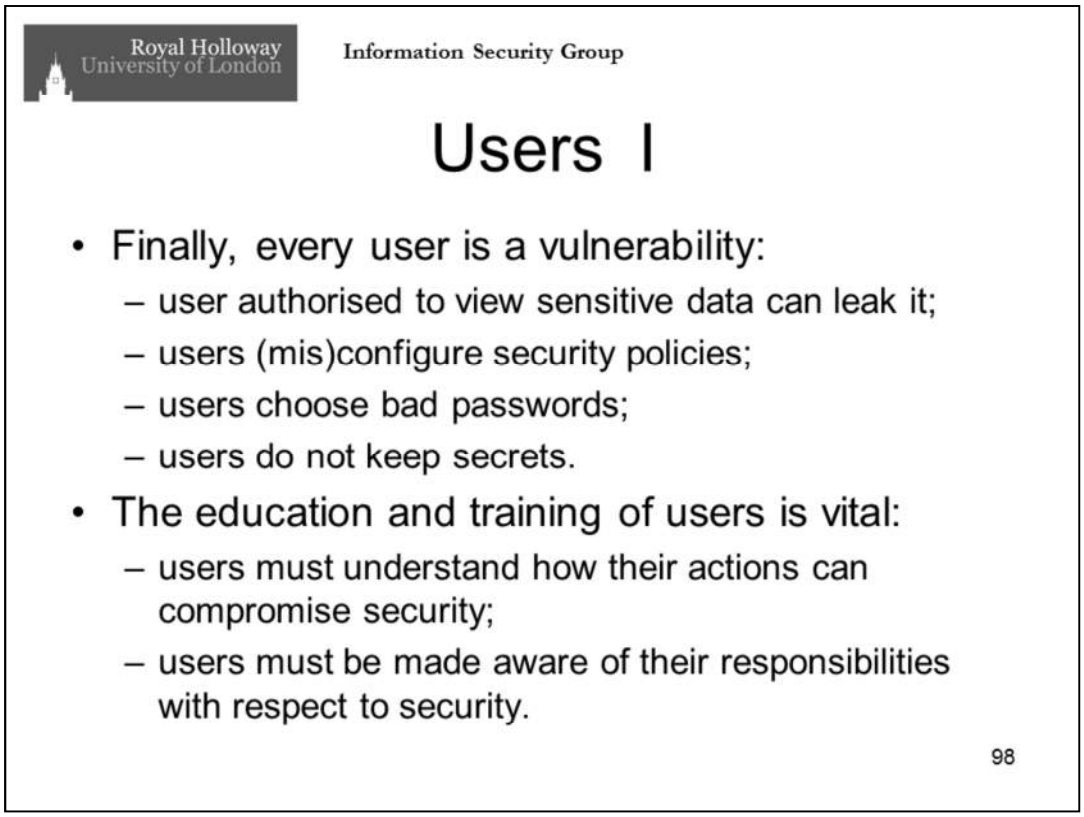


Poor key management:

- cryptographic key management is a complex issue;
- cannot protect long cryptographic keys with potentially weak short passwords.

Customised cryptography:

- it is not normally a good idea to use customised cryptography, except in a large organisation with a dedicated cryptographic team;
- it is surprising how often developers (falsely) believe that they can easily design their own cryptographic primitives, and by doing so they will be developing a more secure system than if they adopt standardised techniques.



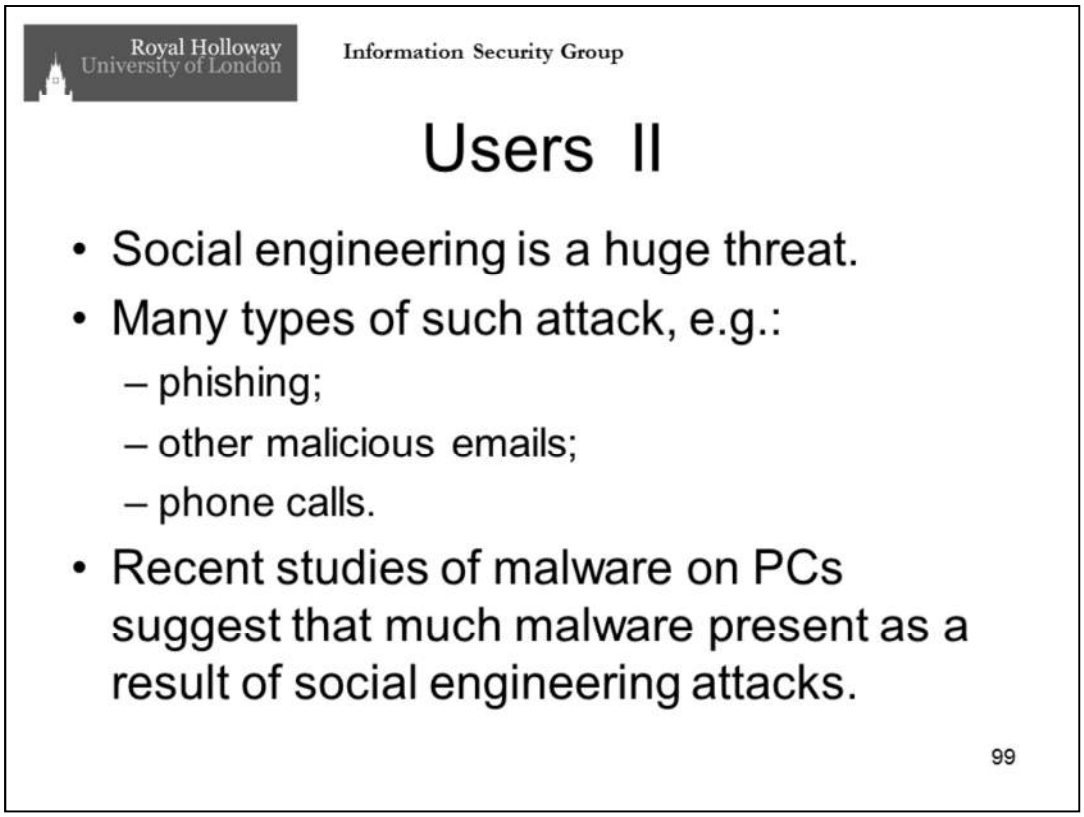
Finally, every user represents a vulnerability:

- any user authorised to view data to which access is restricted can leak that data;
- users (mis)configure security policies;
- users choose bad passwords;
- users do not keep secrets.

The education and training of users is vital:

- users must understand how their actions can compromise security;
- users must be made aware of their responsibilities with respect to security.

Of course, there is a limit to what can be achieved with education. Unreasonable demands must not be placed on users in the name of security, or they will simply ignore them in order to get their job done. That is, security must be **usable**.



To conclude, it is important to realise that so called **social engineering** is a huge threat. Wikipedia defines social engineering as the art of manipulating people into performing actions harmful to themselves or divulging confidential information.

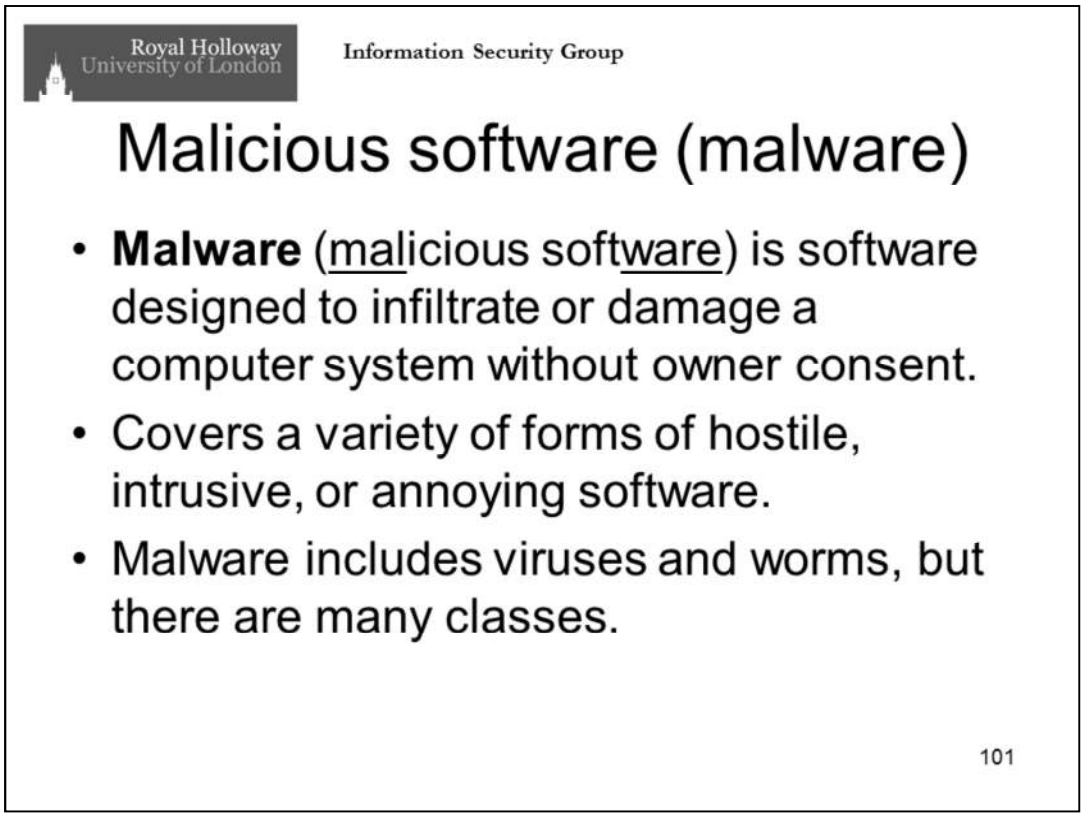
There are many types of social engineering attack, including:

- phishing;
- other malicious emails;
- malicious phone calls (e.g. persuading an innocent user to do damage top their own machine).

Many more examples are described on the Wikipedia page devoted to the subject.

Recent studies of malware on PCs suggest that a significant proportion (most) malware is present as a result of social engineering attacks, not because of any vulnerabilities in the system itself.

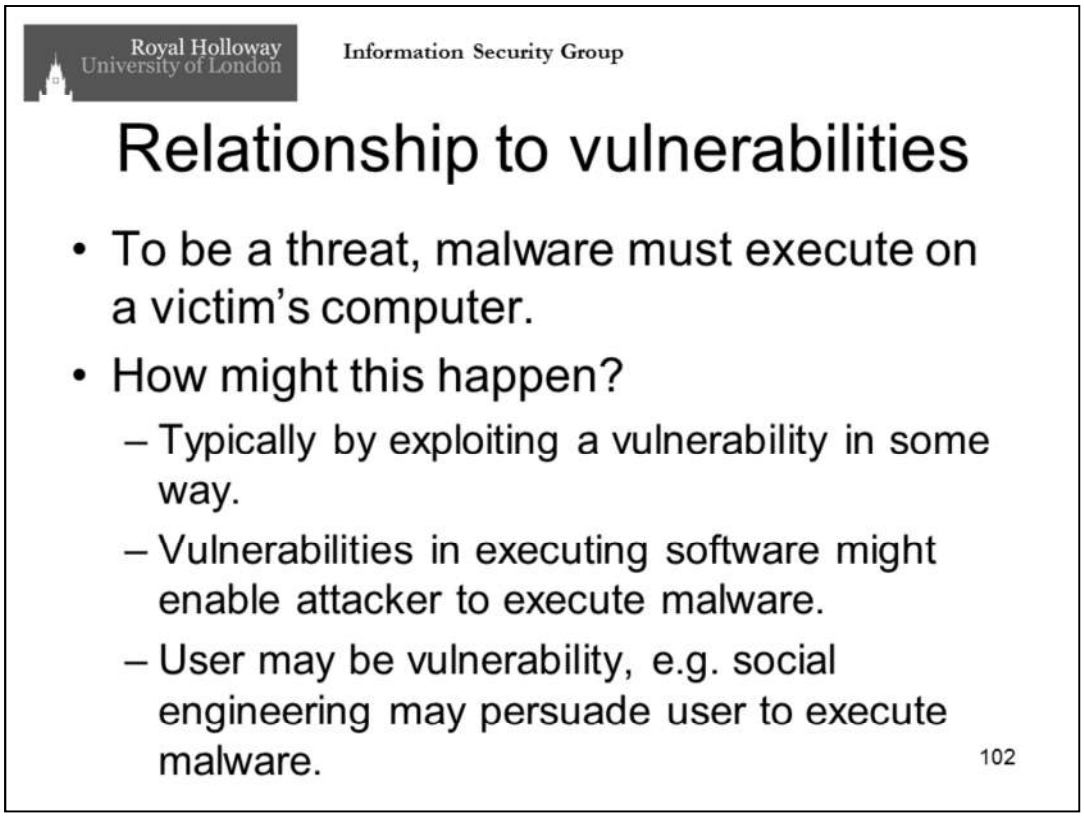
We conclude this part of the course by discussing malware. Malware includes both software designed to exploit vulnerabilities in systems, as well as software which is installed on a machine (and performs harmful acts) as a result of the successful exploitation of a vulnerability.



Malware (malicious software) is software designed to infiltrate or damage a computer system without the owner's consent. The word is used to cover a wide variety of forms of hostile, intrusive, or annoying software.

Malware includes viruses and worms, but there are many other categories of malware.

Initial infection by a worm or virus is often followed by the installation of additional malware such as rootkits, trojan horse programs, adware and spyware.



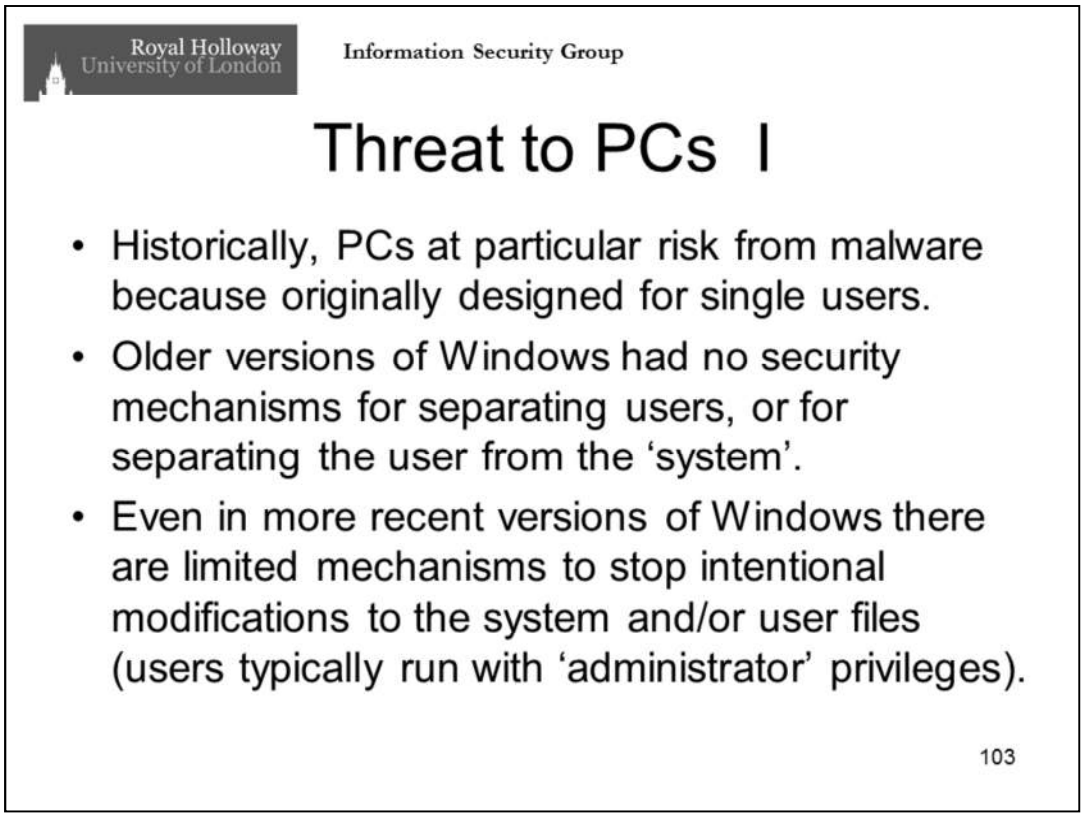
To be a threat, malware must execute on a victim's computer. There are many ways in which this might happen. Typically it involves the exploitation of a vulnerability. Examples include the following.

• Vulnerabilities in executing software might enable an attacker to execute malware. For example, a 'data file' opened by a word processor might contain a malicious code fragment, which is executed as a result of poor data validation by the word processor, or through use of an unprotected macro execution facility.

• Depending on how the user's computer is configured, malware contained on a portable memory device may be automatically executed when connected to a victim's computer.

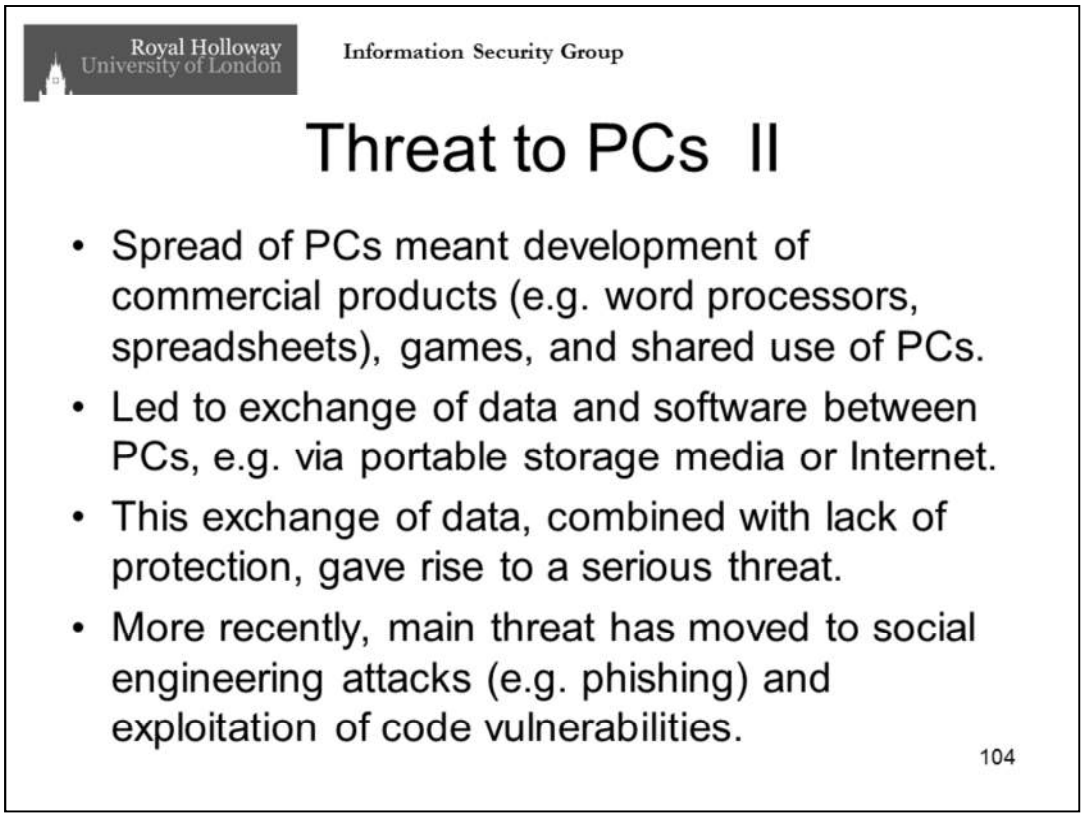
• The user may be the vulnerability him/herself; for example, the user might be persuaded through a carefully crafted email or web page to download and execute a piece of malware (thinking it is something else).

• Some malware, notably viruses and worms (which we look at a little later in this part of the course), propagates itself automatically from computer to computer. However, malware does not have to be self-propagating to be very dangerous.



Historically, PCs have been at particular risk from malicious code because they were originally designed for single users. PC operating systems such as Windows originally contained no security mechanisms for separating users, or for separating the user from the 'system'. Neither did they incorporate any security mechanisms to stop intentional modifications to the system and/or user files.

Even in more recent versions of Windows there are limited security mechanisms to stop intentional modifications to the system and/or user files (users typically run with 'administrator' privileges). Note that Vista, Windows 7 and Windows 8 contain measures designed to try to address this – steps requiring administrator privileges require explicit user authorisation.

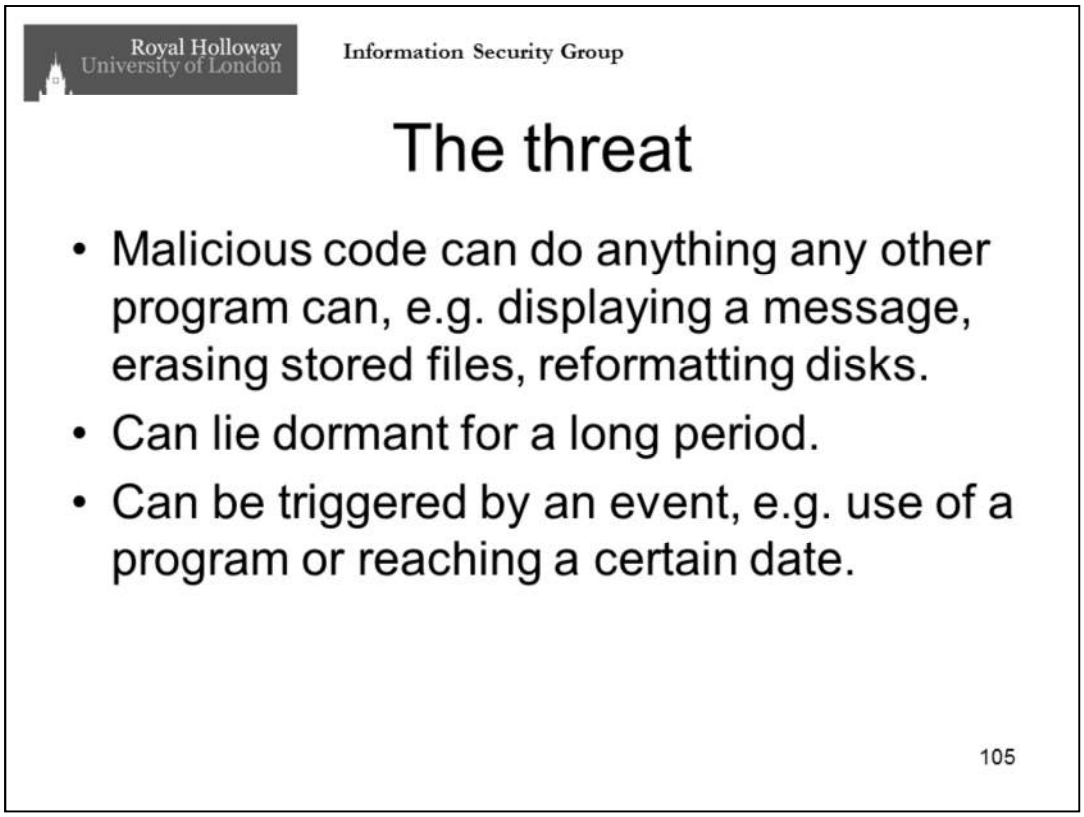


The spread of PCs has meant the development of a variety of commercial products (e.g. word processors, spreadsheets), as well as games. Also single PCs are shared by multiple users.

The growth of commercial applications has led to the widespread exchange of data between PCs, e.g. via portable storage media or the Internet.

These exchanges of data and software, combined with the lack of protection in PC operating systems, originally gave rise to the malware threat (historically in the form of viruses).

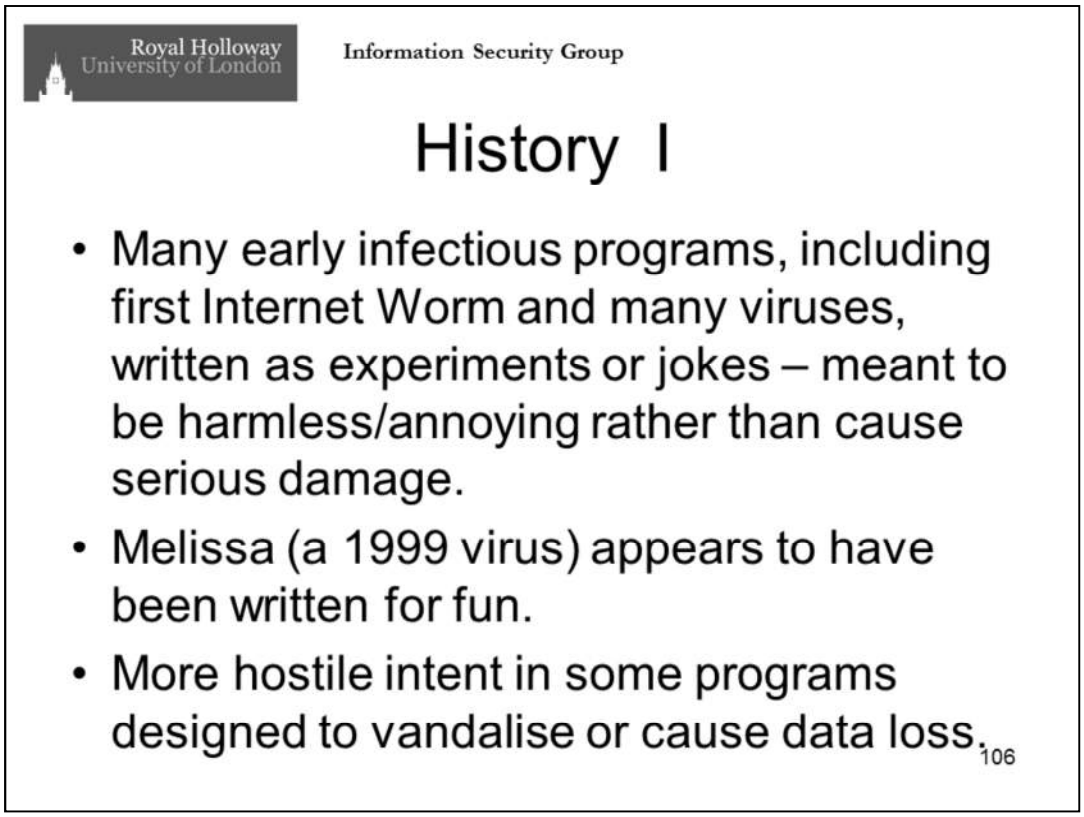
More recently, the main threat has moved to social engineering attacks (e.g. phishing) and exploitation of code vulnerabilities.



Malicious code can do anything any other program can, e.g. displaying a message, erasing stored files, and/or reformatting disks.

Malicious code can lie dormant within the PC for a long period. Its effects can be triggered by various types of event, e.g. the use of some program or reaching a certain date.

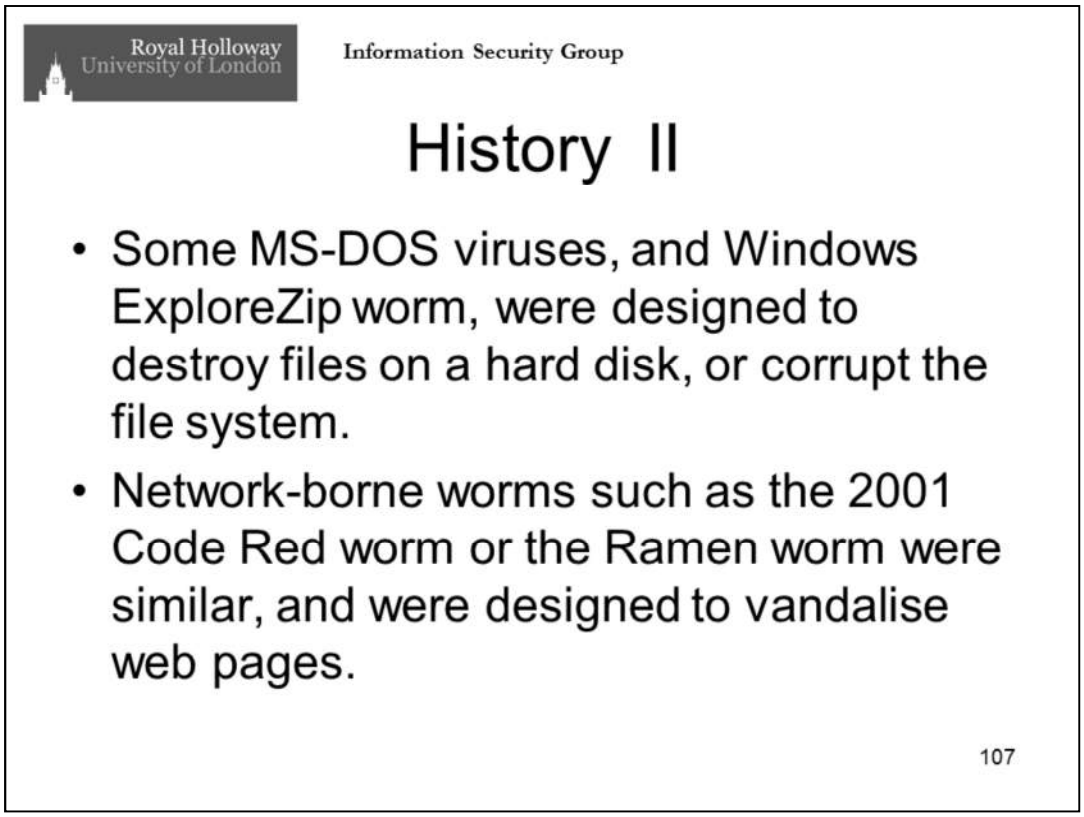
What makes the types of malicious code known as *viruses* and *worms* particularly dangerous is their ability to replicate and spread.



Many early infectious programs, including the first Internet Worm and many of the first viruses seen in the wild, were written as experiments or jokes – they were intended to be harmless or annoying, rather than cause serious damage.

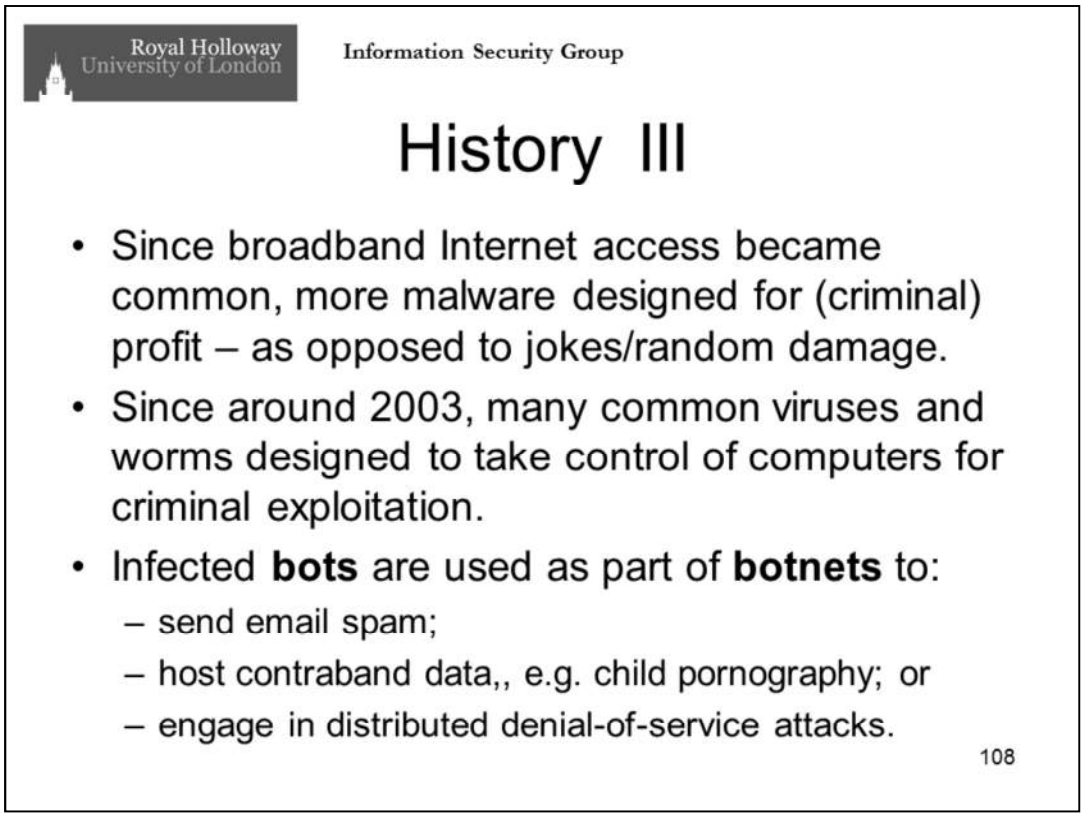
Viruses such as Melissa (from 1999) appear to have been written for fun.

There was, nevertheless, slightly more hostile intent in some programs that were designed to vandalise or cause data loss.



Even in the 'early days' of malware, some MS-DOS viruses and worms (e.g. Windows ExploreZip), were designed to destroy files on a hard disk, or to corrupt the file system by writing junk data.

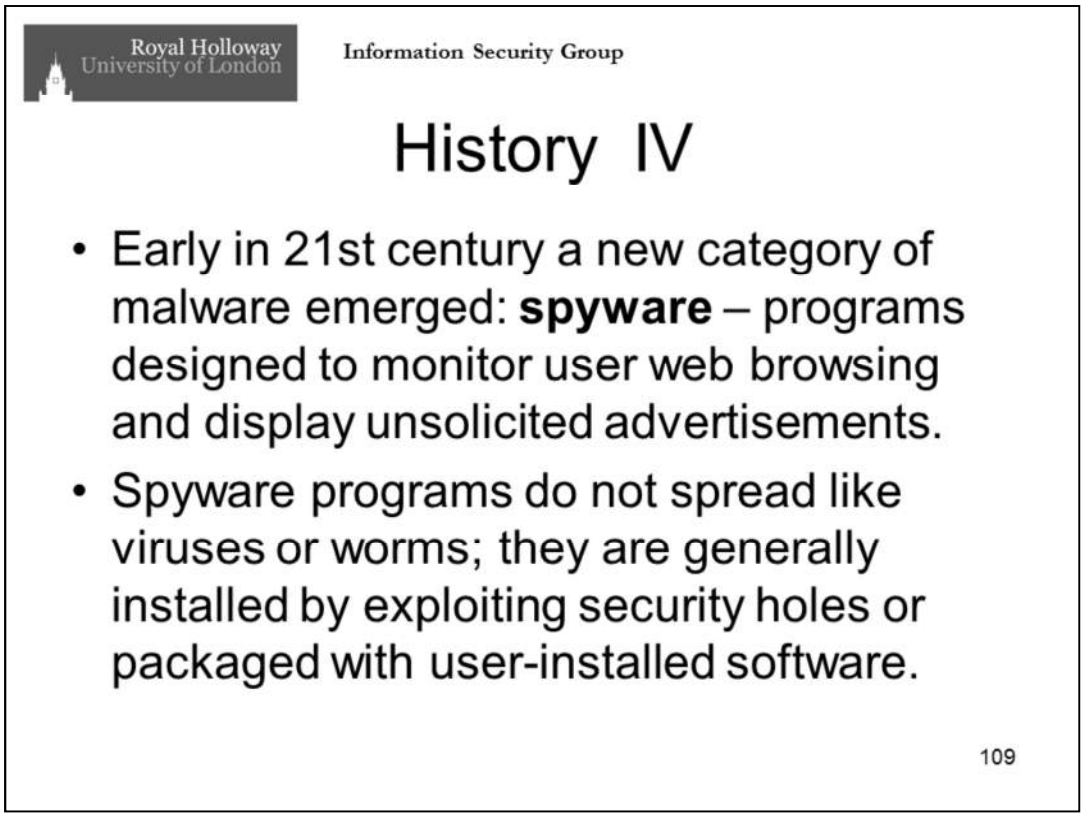
Network-borne worms such as the 2001 Code Red worm or the Ramen worm were similar, and were designed to vandalise web pages.



Since broadband Internet access has become common, malware has been increasingly designed for (criminal) profit, as opposed to previous malware which was mainly designed either just to prove a point for fun or to cause random damage.

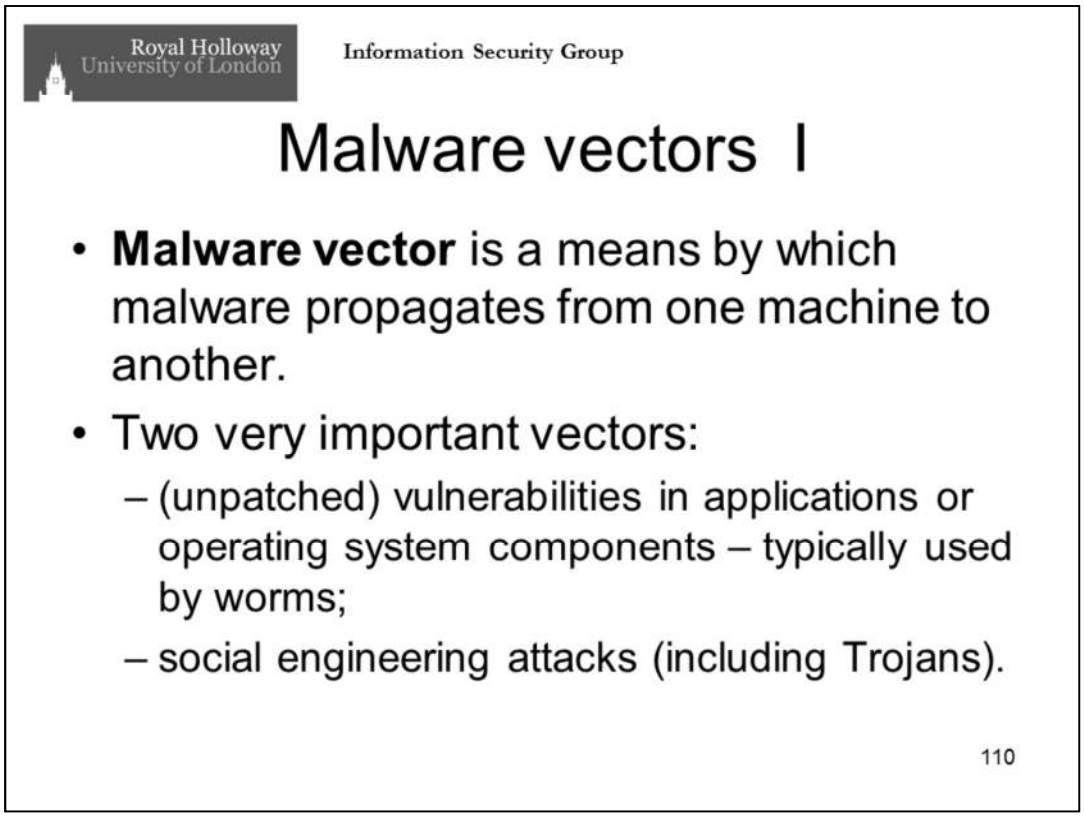
Since around 2003, many of the commonly occurring viruses and worms have been designed to take control of computers for criminal exploitation.

Infected **bots**, typically making up a centrally controlled **botnet**, can be used for a variety of purposes, including to send email spam, to host contraband data such as child pornography, or to engage in distributed denial-of-service attacks, e.g. as a form of extortion.



Early in the 21st century another category of malware emerged, known as spyware, i.e. programs designed to monitor user web browsing and display unsolicited advertisements.

Spyware programs are typically not infectious, like viruses, or self-replicating, like worms; they are generally installed by exploiting security holes or are packaged with user-installed software (one prominent early example was circulated by Kazaa).



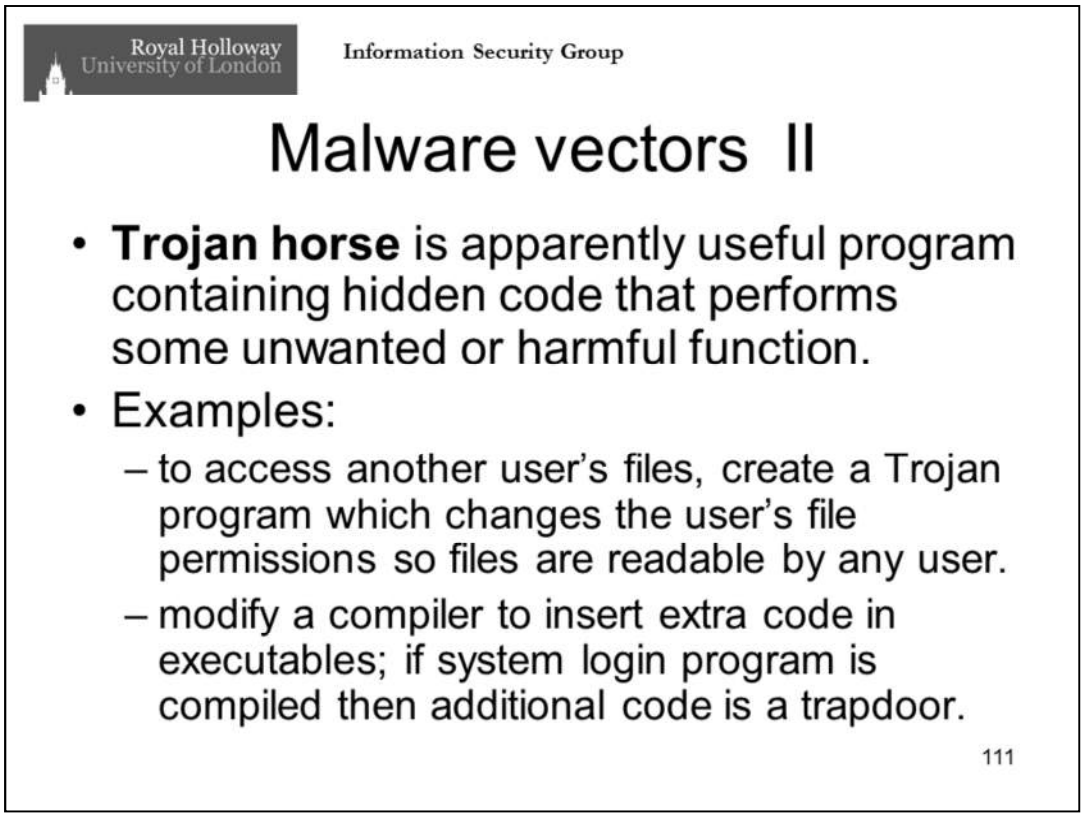
A **malware vector** is a means by which malware propagates from one machine to another.

There are two particularly important vectors:

- (unpatched) vulnerabilities in applications or operating system components;
- social engineering attacks (including Trojans – see next slide).

The former vector is typically used by worms in propagating across networks between machines.

Viruses propagate in a range of ways, including through social engineering, and through use of unintended functionality (see next slide), as well as sometimes using vulnerabilities.

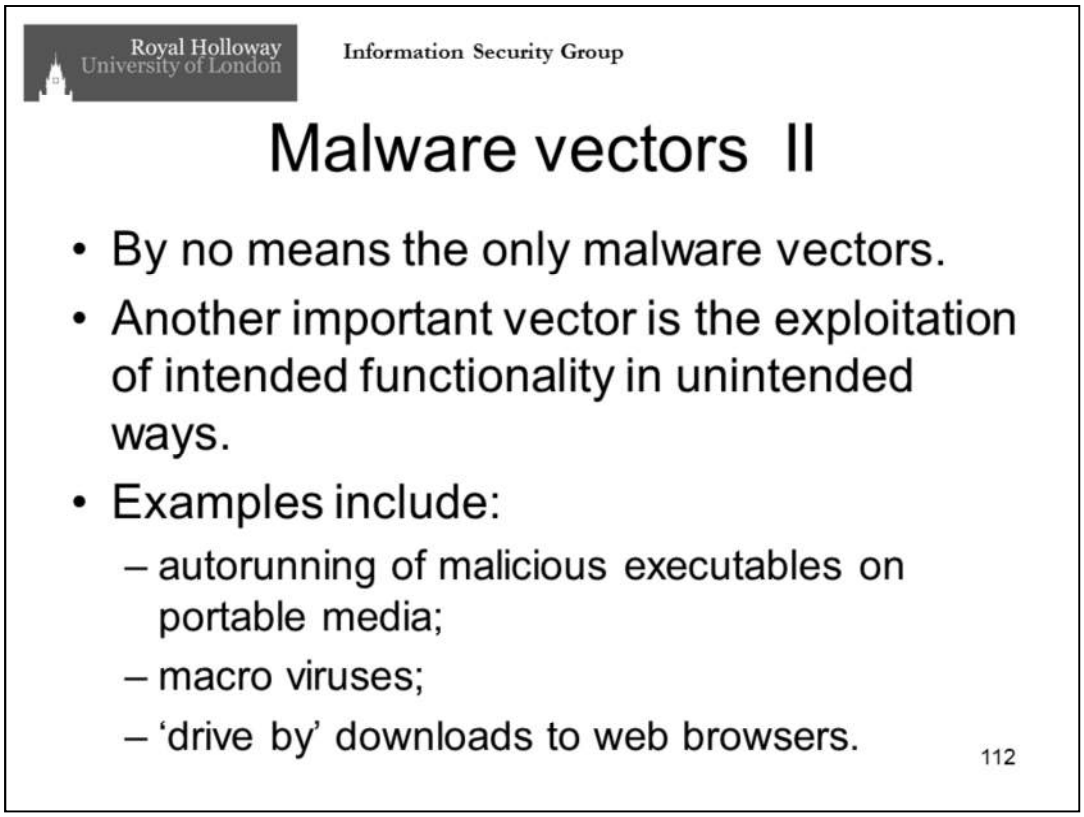


A *Trojan horse* is an apparently useful program containing hidden code that, when invoked, performs some unwanted or harmful function. Examples include:

• in order to gain access to another user's files on a shared system, the attacker creates a Trojan program which, when executed, changes the invoking user's file permissions so that files are readable by any user.

• modify a compiler so that it inserts additional code into certain files as it compiles them; if the system login program is compiled then the additional code creates a trapdoor.

Trojans can be regarded as a special type of social engineering attack.



Vulnerabilities in software and social engineering attacks (including Trojans) are by no means the only malware vectors.

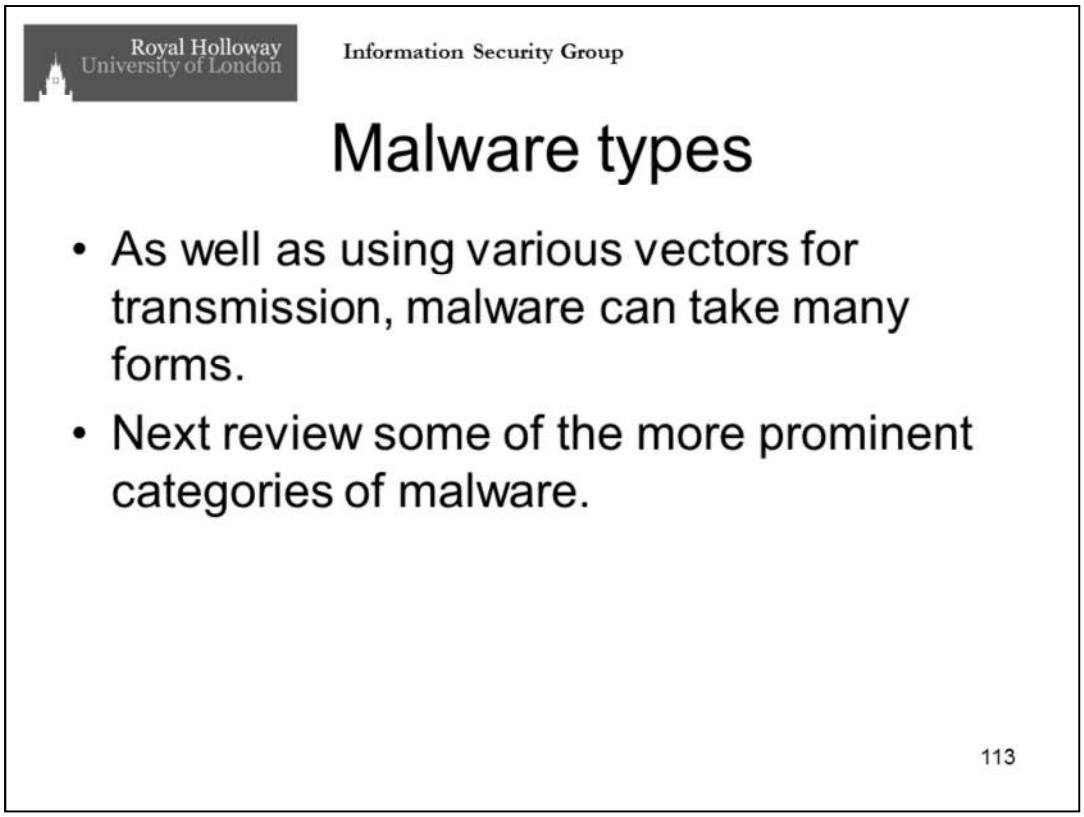
Another important malware vector is the exploitation of intended functionality in unintended ways.

Examples include:

• automatic execution (autorunning) of malicious executables on portable media;

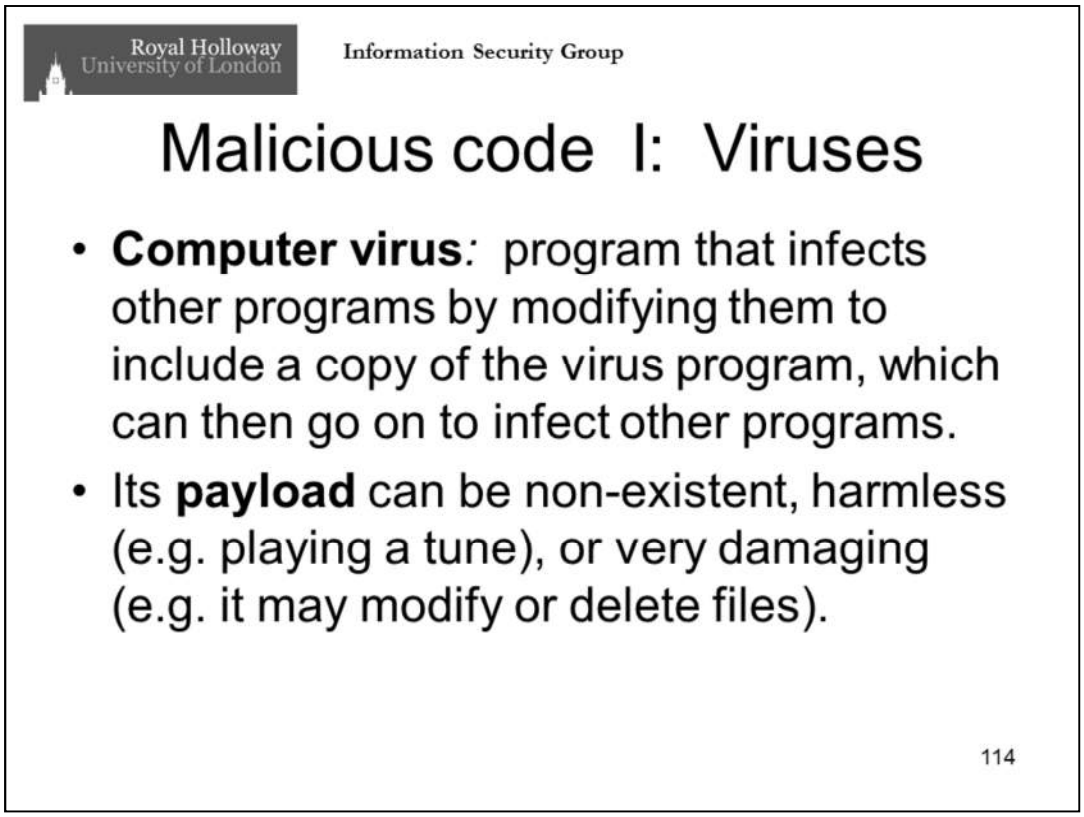
• macro viruses, i.e. viruses containing programs written in a macro language, i.e. a special programming language built into an application such as a word processor or database, intended to be used to automate certain actions;

• 'drive by' downloads to web browsers, i.e. where a user innocently visits a web page which downloads malware to the user's computer; some older browsers would execute such software automatically.



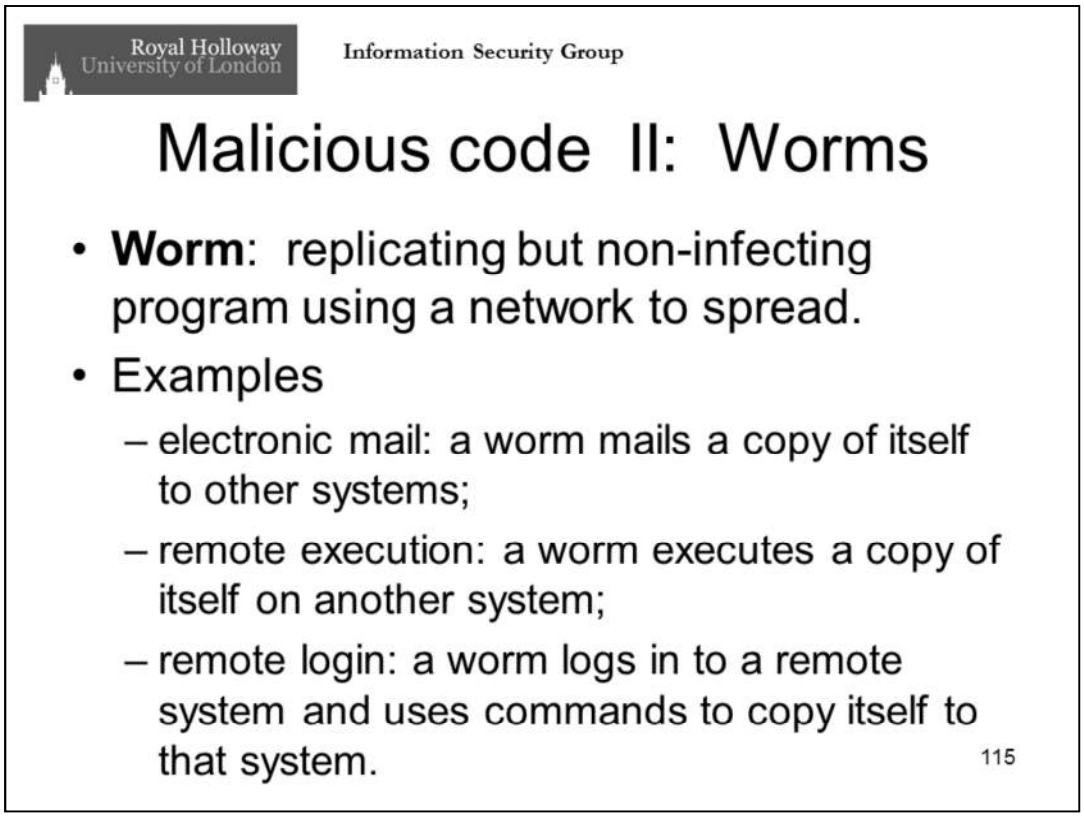
As well as using a variety of vectors for transmission, malware can take many forms.

We next review some of the more prominent categories of malware. Note that malware does not always conveniently fit into one of the categories, and also the categories we present are not mutually exclusive. However, most commonly encountered malware fits in at least one of these categories.



Computer virus: a program that infects other programs by modifying them to include a copy of the virus program, which can then go on to infect other programs.

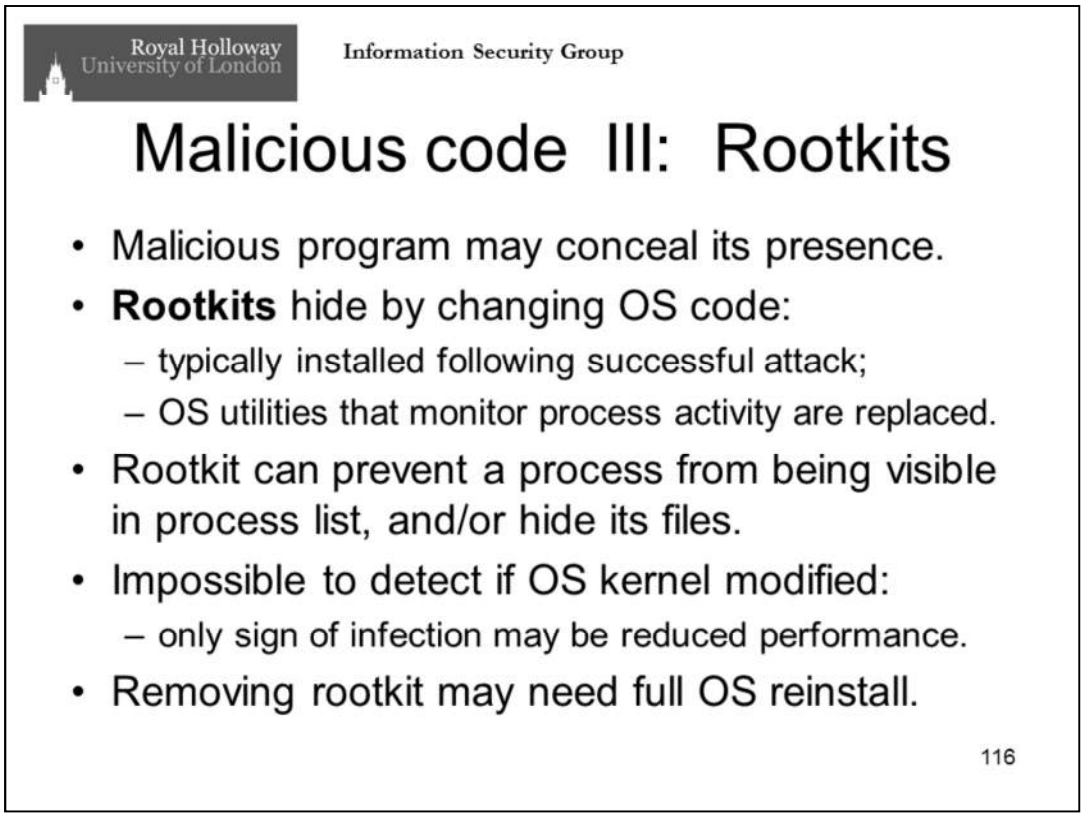
Its **payload** can be non-existent, harmless (e.g. playing a tune), or very damaging (e.g. it may modify or delete files).



A *worm* is a replicating but non-infecting program that uses network connections to spread from system to system.

Examples include:

- electronic mail: a worm mails a copy of itself to other systems;
- remote execution: a worm executes a copy of itself on another system;
- remote login: a worm logs onto a remote system as a user and then uses commands to copy itself from one system to the other.



Once a malicious program has been installed on a system, it is often useful to the creator if it stays concealed.

Rootkits deliberately hide themselves by subverting or replacing operating system code:

• typically installed following successful attack (exploitation of a vulnerability), e.g. via a Worm;

• OS utilities that monitor process activity are replaced with programs that hide malicious activity.

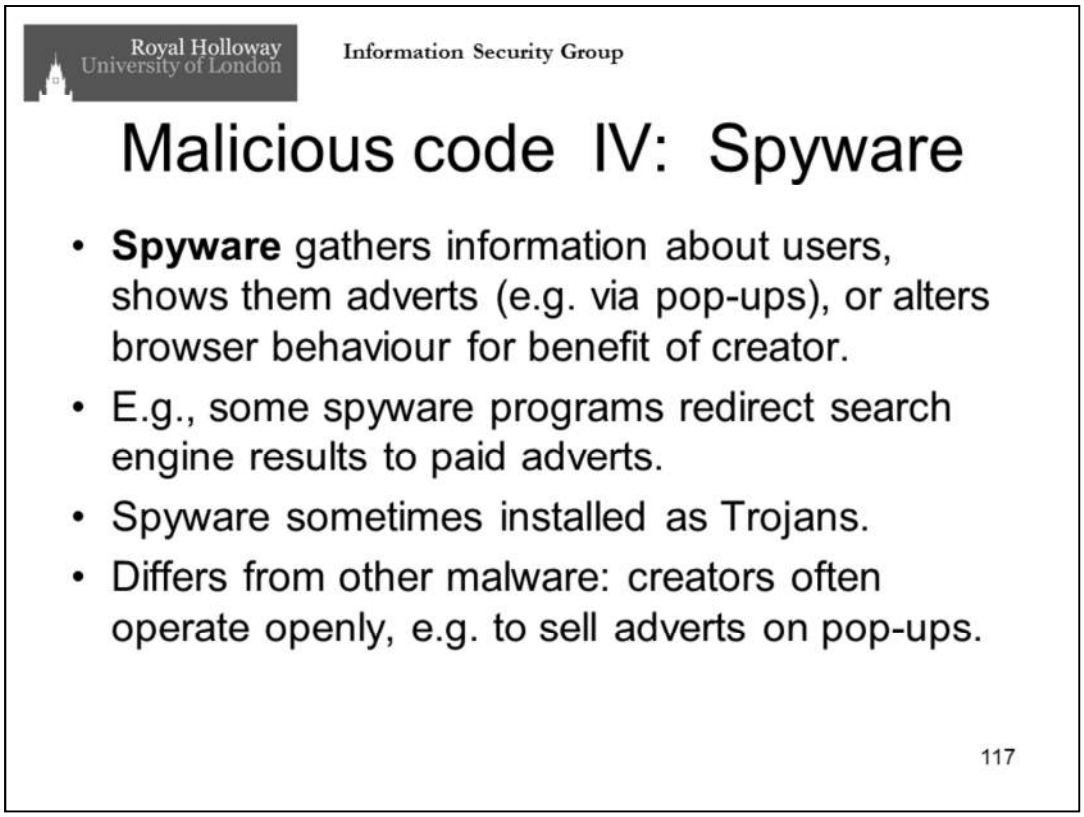
Rootkits can prevent a malicious process from being visible in the system process list, or keep its files from being read.

It may be impossible to detect a rootkit if the OS kernel has been partially replaced:

• only sign of rootkit infection may be reduced performance.

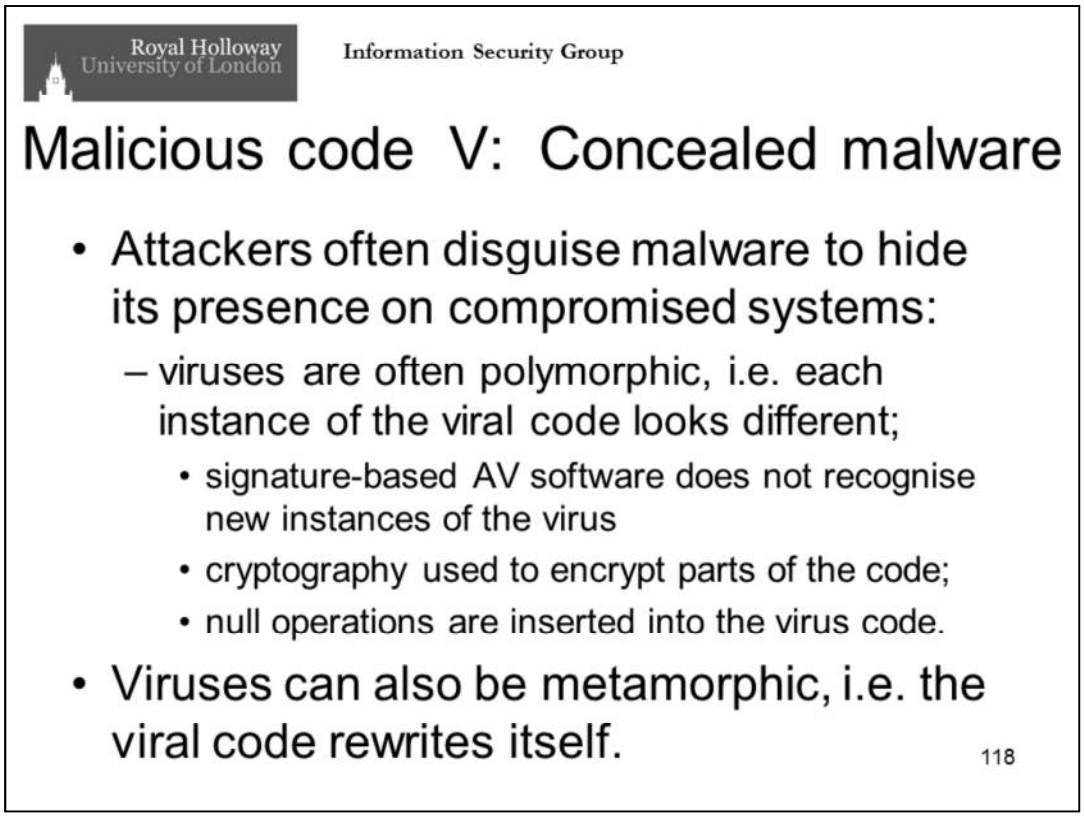
Removing a rootkit often requires a complete reinstall of the OS.

Originally, a rootkit was a set of tools installed by a human attacker on a Unix system where the attacker had gained administrator (root) access. Today, the term is used more generally for concealment routines in a malicious program.



Over the last ten-fifteen years or so, one of the most costly forms of malware in terms of time and money spent in recovery has been the broad category known as spyware. Spyware programs are commercially produced for the purpose of gathering information about computer users, showing them advertisements (e.g. via pop-ups), or altering web-browser behaviour for the financial benefit of the spyware creator. For example, some spyware programs redirect search engine results to paid advertisements

Spyware programs are sometimes installed as Trojan horses. They differ from other types of malware that their creators often present themselves openly as businesses, e.g. by selling advertising on the pop-ups created by the malware. Most such programs present the user with an end user agreement which supposedly protects the creator from prosecution under computer abuse laws - however, the precise legality of these agreements remains to be determined.

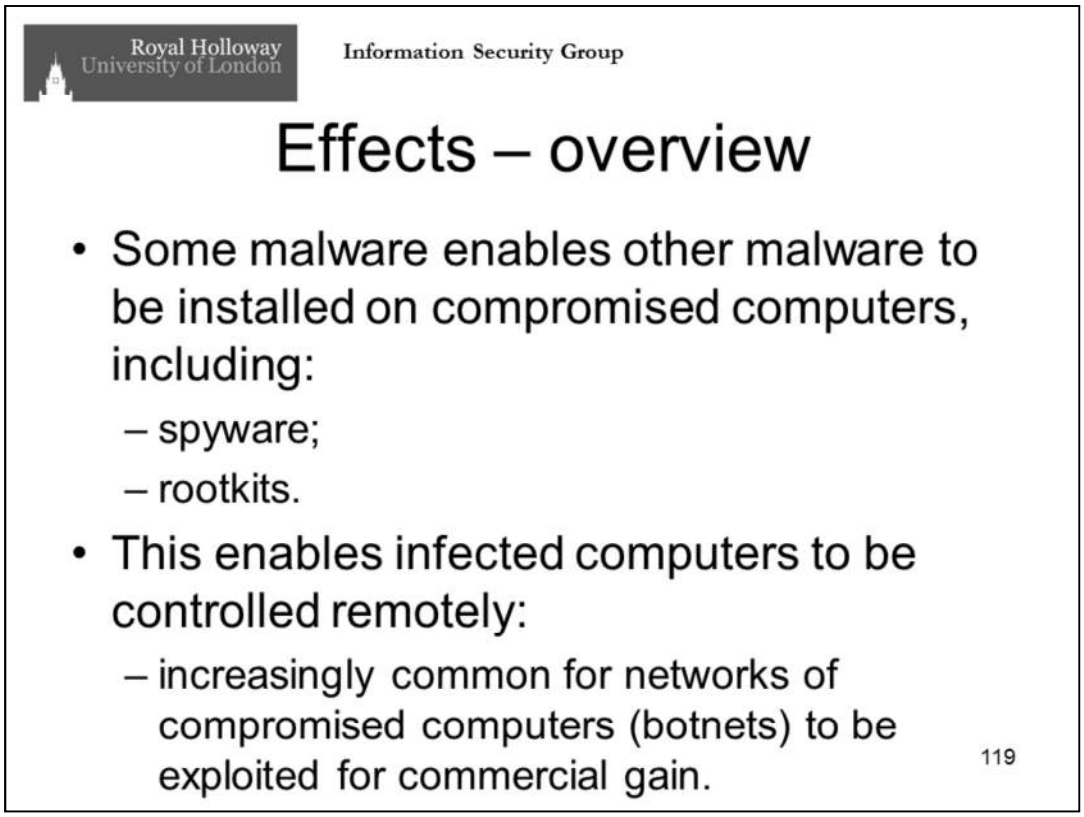


Attackers often disguise malware to hide its presence on compromised systems:

- as alluded to previously, viruses are often polymorphic, i.e. each instance of the viral code looks different;
  - signature-based AV software does not recognise new instances of the virus
  - cryptography used to encrypt parts of the code;
  - null operations are inserted into the virus code.

Viruses can also be metamorphic, i.e. the viral code rewrites itself.

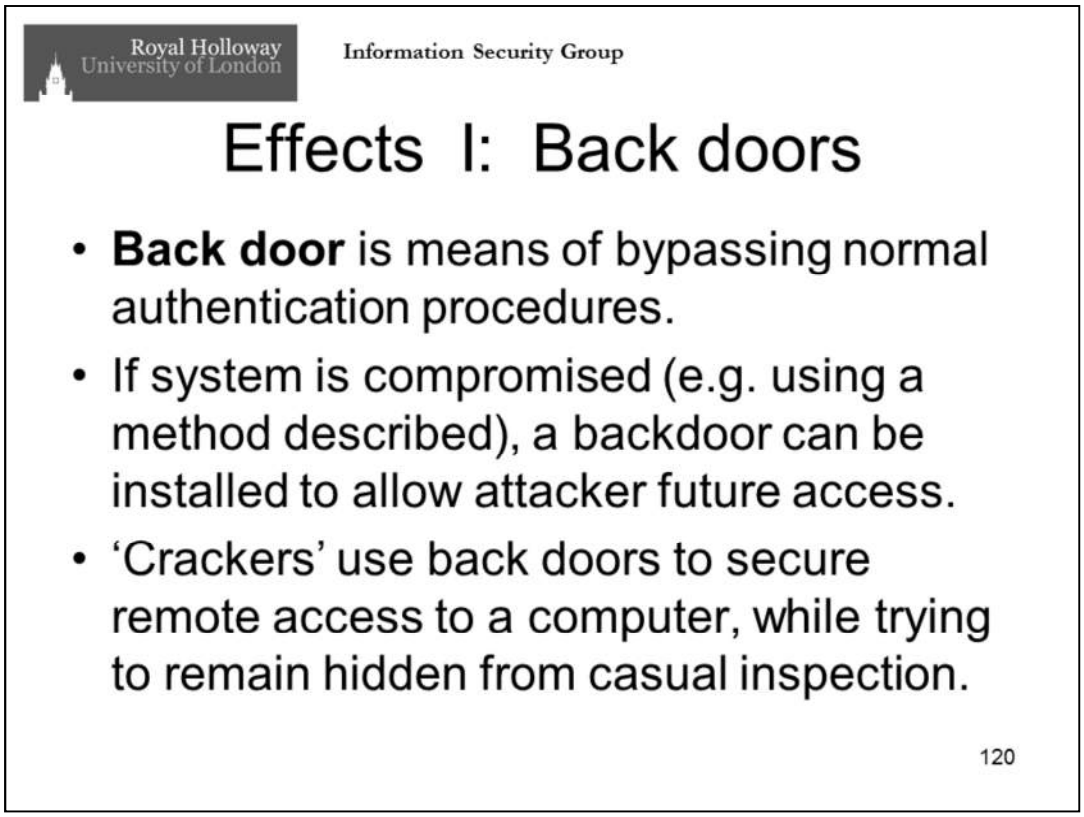
Of course, rootkits (as mentioned previously) also conceal their presence.



Some malware has as its main function enabling other malware to be installed on compromised computers, including spyware and rootkits.

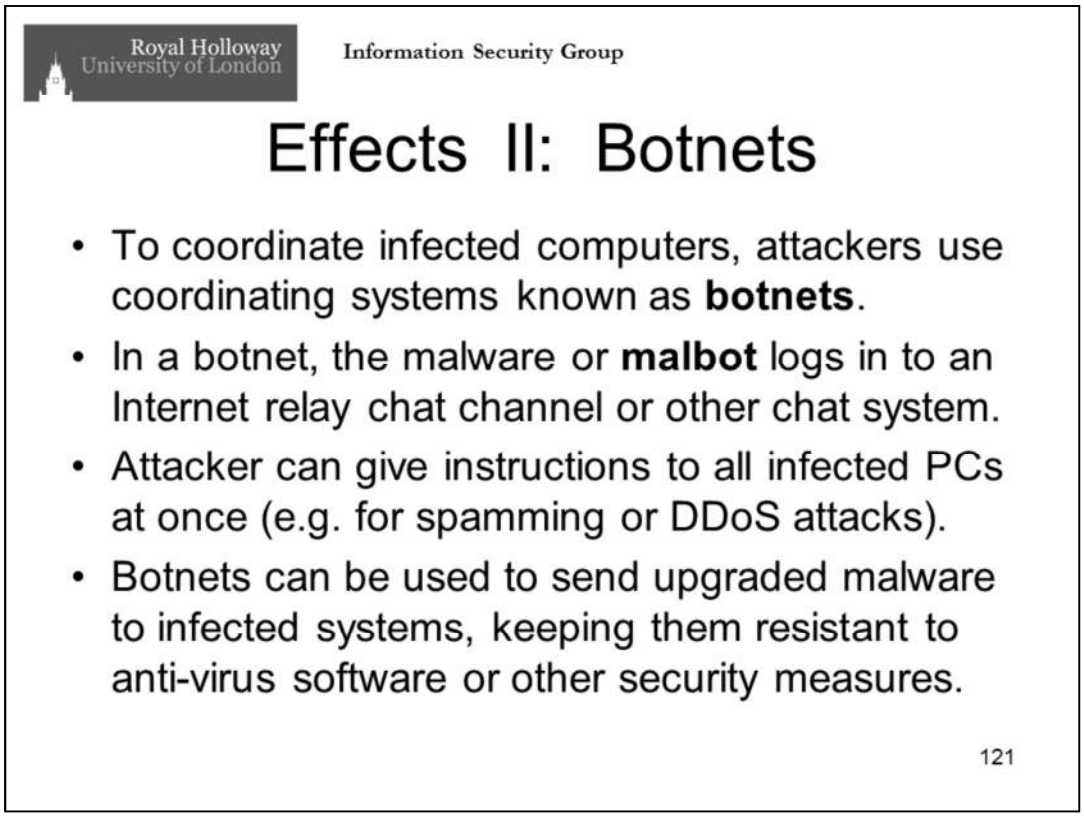
The installed malware often enables infected computers to be controlled remotely:

• it has become common for criminal entities to exploit networks of compromised computers (botnets) for commercial gain.

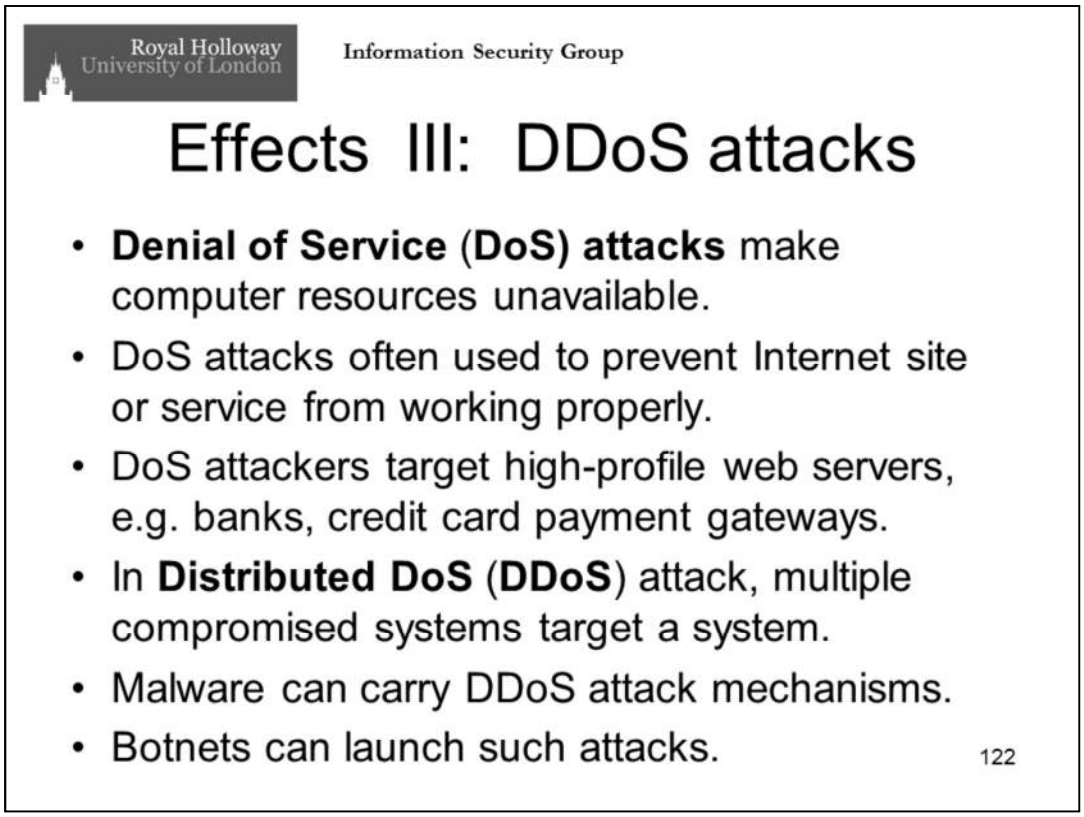


A back door is a method of bypassing normal authentication procedures. Once a system has been compromised (by one of the methods described, or in some other way), one or more backdoors could be installed, in order to allow the attacker access in the future.

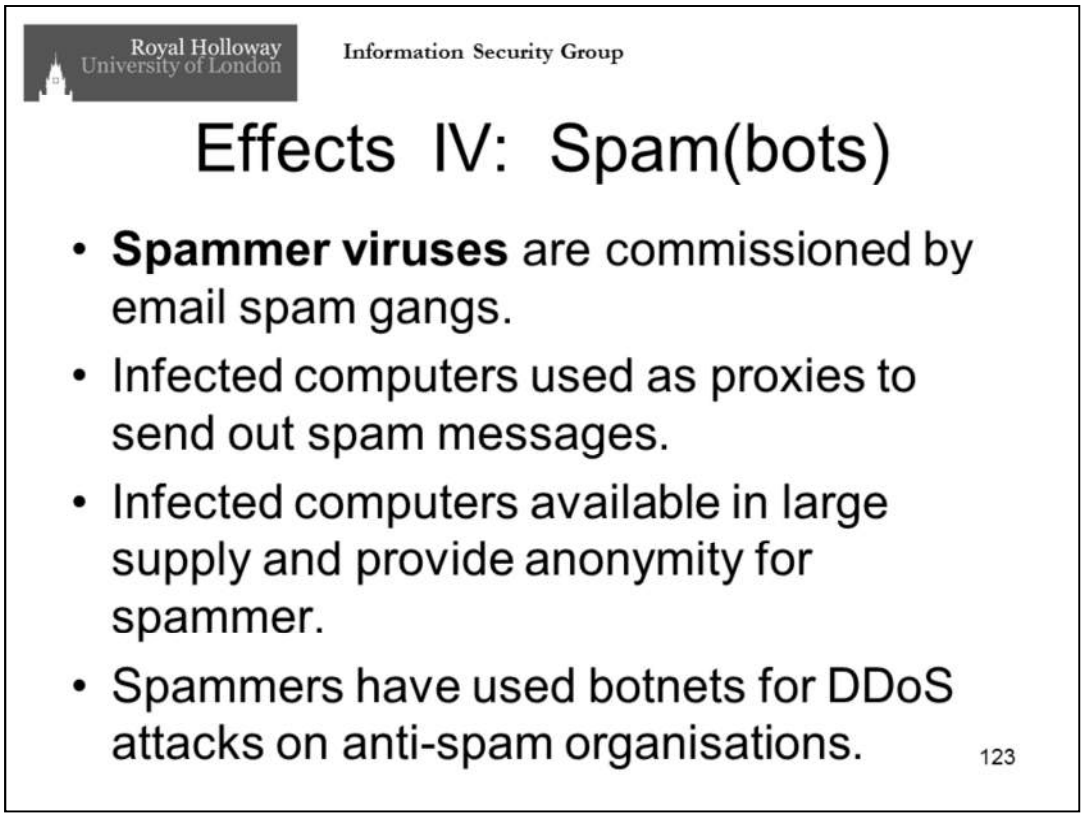
So called 'crackers' typically use backdoors to secure remote access to a computer, while attempting to remain hidden from casual inspection. To install back doors, these crackers may use Trojans, worms, etc.



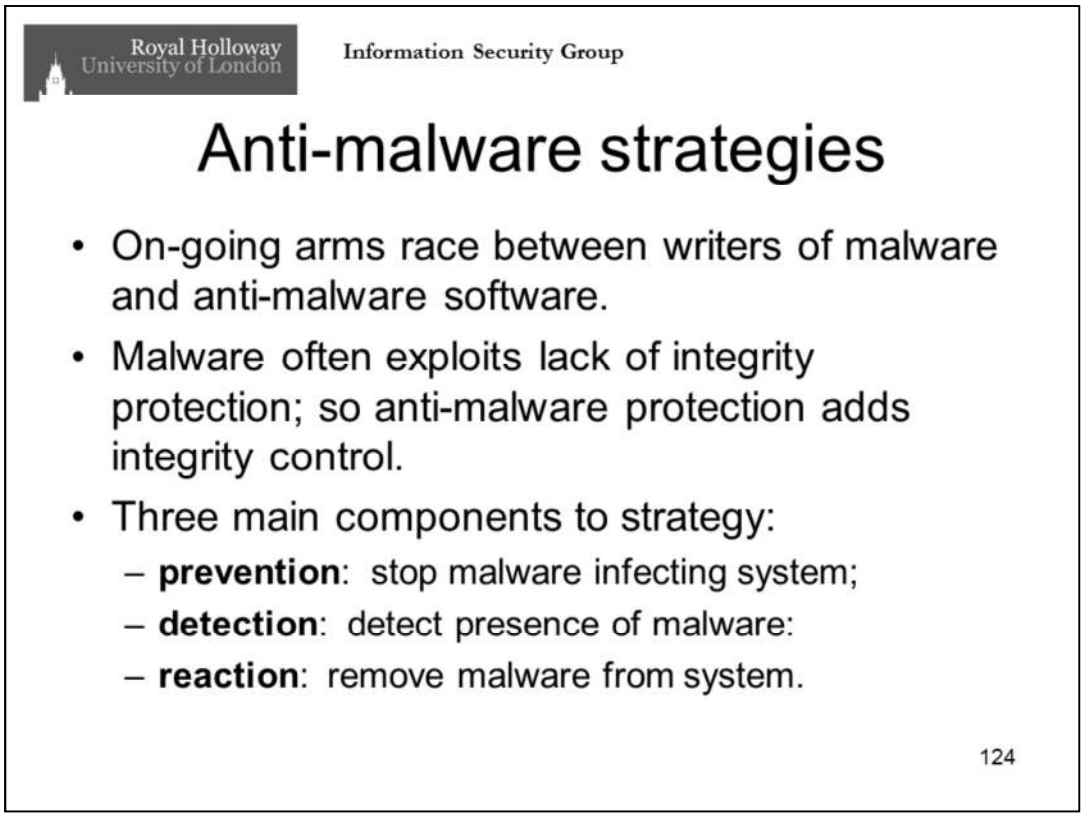
In order to coordinate the activity of many infected computers, attackers have used coordinating systems known as **botnets**. In a botnet, the malware or **malbot** (or just **bot**) logs in to an Internet relay chat channel or other chat system. The attacker can then give instructions to all the infected systems simultaneously, e.g. to conduct DDoS attacks or send email spam. Botnets can also be used to push upgraded malware to the infected systems, keeping them resistant to anti-virus software or other security measures.



A **Denial of Service** (**DoS) attack** is an attempt to make a computer resource unavailable to its authorised users. Although the means to conduct a DoS attack, and the motives for and targets of a DoS attack may vary, it generally involves a concerted, malevolent effort to prevent an Internet site or service from functioning properly, temporarily or indefinitely. Perpetrators of DoS attacks typically target sites or services hosted on high-profile web servers such as banks, credit card payment gateways and even DNS root servers. A **Distributed Denial of Service** (**DDoS**) **attack** occurs when multiple compromised systems flood the bandwidth or resources of a targeted system, usually one or more web servers. Malware can carry DDoS attack mechanisms; one of the more well known examples of this was MyDoom – its DoS mechanism was triggered on a specific date and time. This type of DDoS involved hardcoding the target IP address prior to release of the malware and no further interaction was necessary to launch the attack.



Spammer viruses, such as the Sobig and MyDoom virus families, are commissioned by email spam gangs. Infected computers are used as proxies to send out spam messages. The advantage to spammers of using infected computers is that they are available in large supply (thanks to the virus) and they provide anonymity, protecting the spammer from prosecution. Spammers have also used infected PCs to target anti-spam organizations with DDoS attacks.

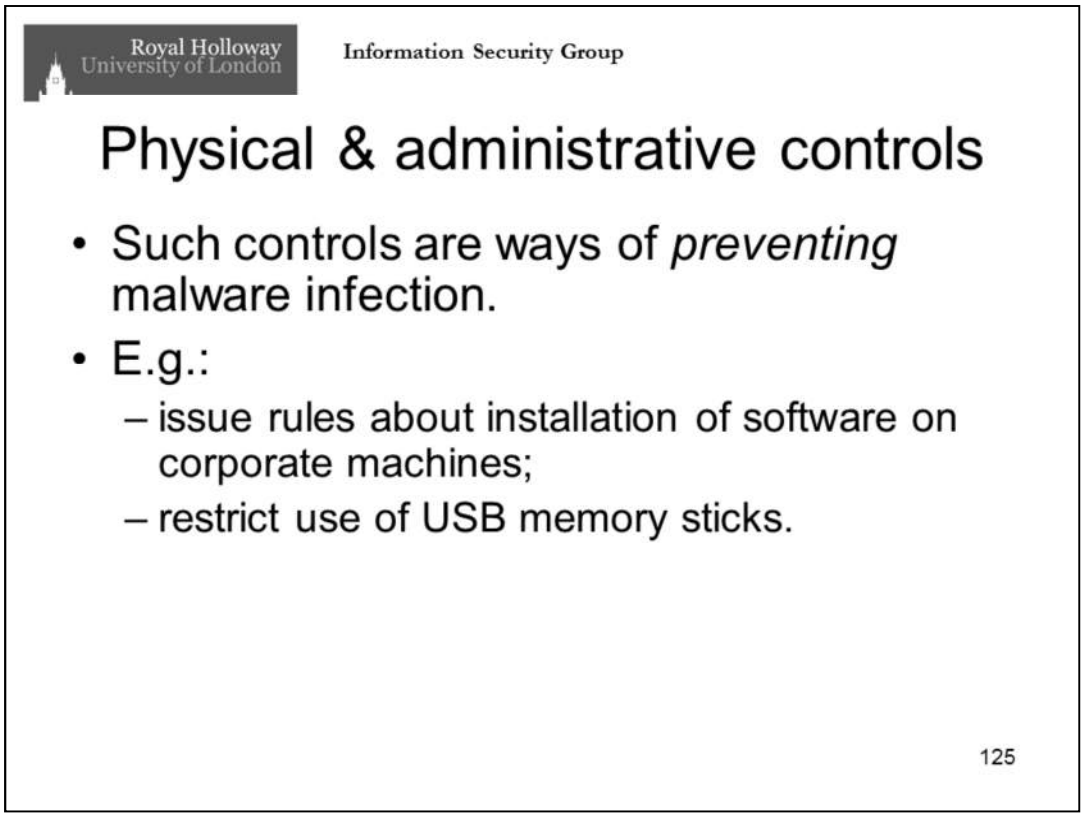


We now consider how one should approach the problem of dealing with the threat posed by malware. First note that viruses exploit the lack of integrity protection in a system, and hence an anti-malware protection strategy involves adding integrity control to a system.

There are three main components to a complete anti-malware strategy:

- *Prevention*: i.e. stopping malware infecting the system;
- *Detection*: i.e. detection of the presence of malware in the system;
- *Reaction*: i.e. removal of malware from the system.

Historically, the main issues were viruses and worms, but the emphasis has changed over the last decade.

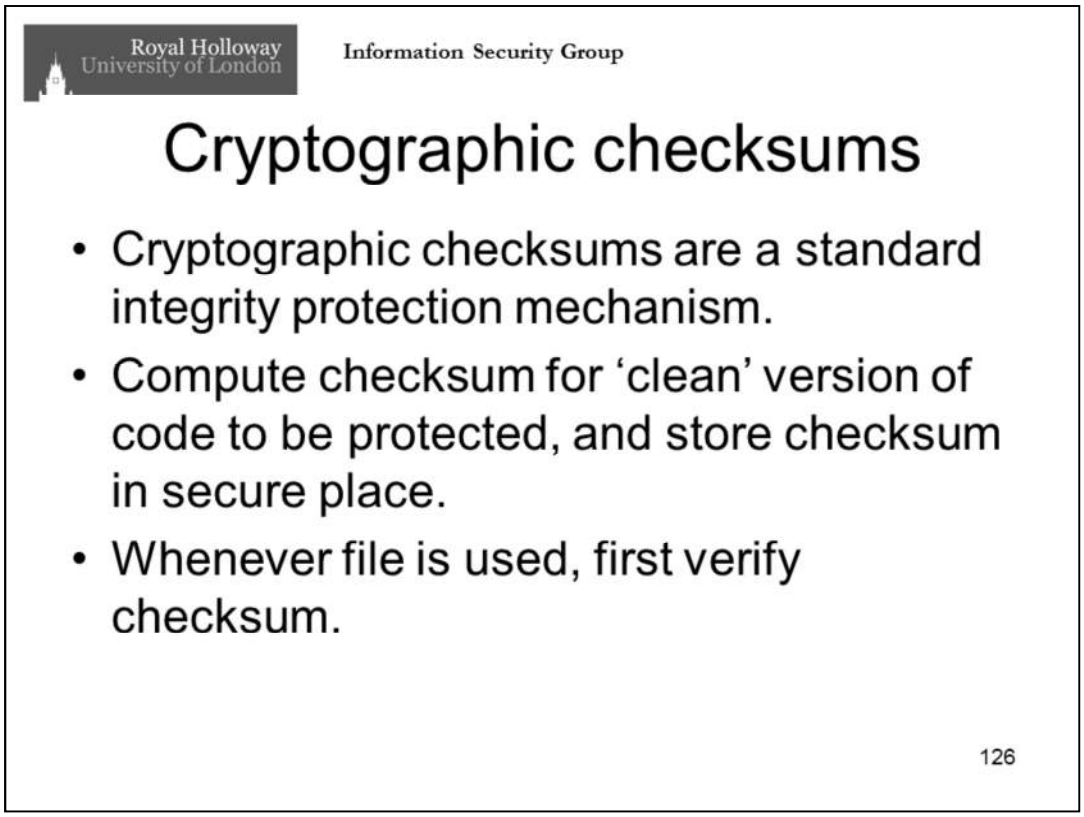


Physical and administrative controls can be effective ways of preventing malware infection, including by viruses. There are many examples of suitable controls of this type. Examples include the following.

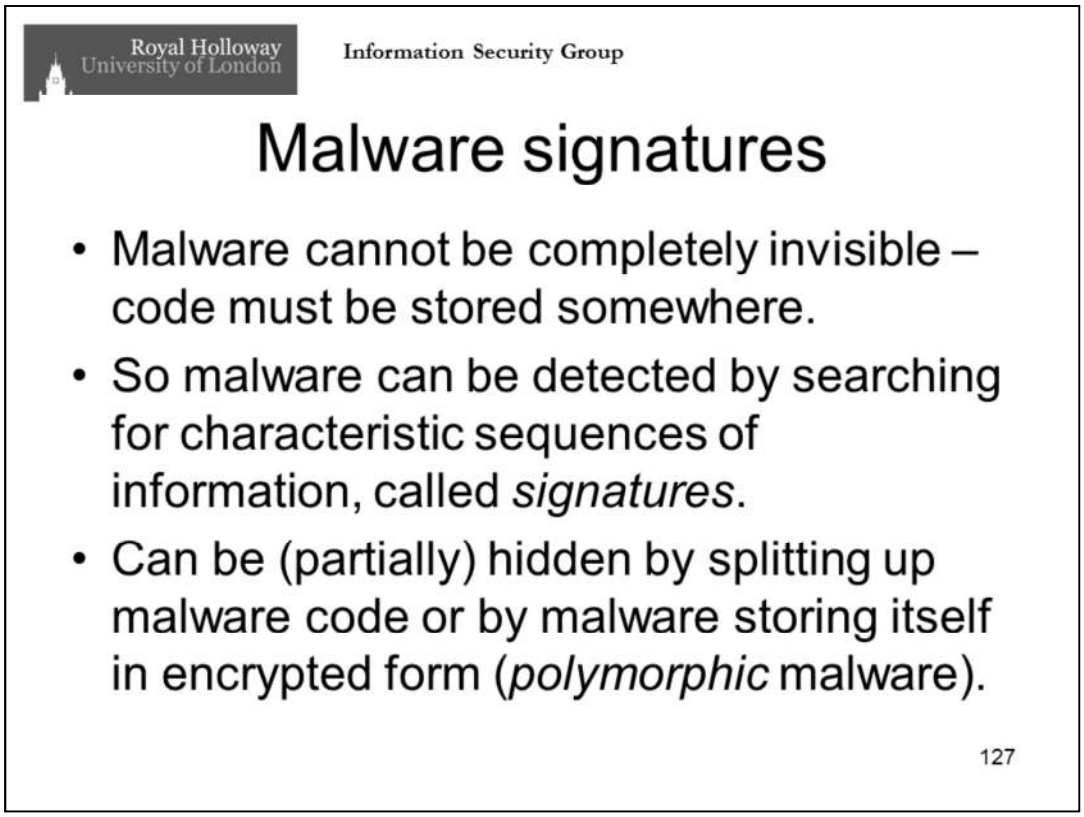
• Many companies will have a policy forbidding use of any software which has not been explicitly approved by the company's IT or security department. Thus all software must be procured by official means, minimising the risk of infection by malware.

• Restrict use of USB memory sticks for data transfer. USB memory sticks have become a very serious malware vector (particularly when combined with the autorun feature).

However, this all relies on user cooperation and understanding which is difficult to guarantee, particularly when facing threats like spear phishing.



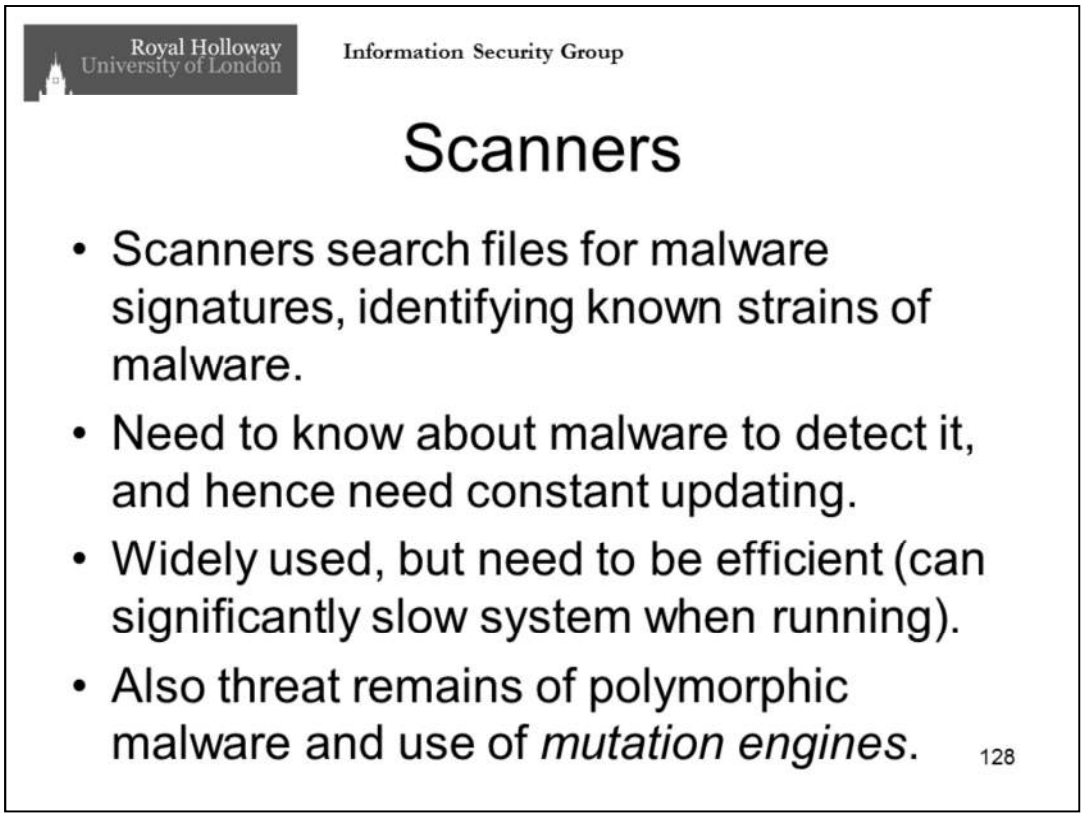
Cryptographic checksums are a standard integrity protection mechanism. To use it to protect against malware infection, compute the checksum for a 'clean' version of the code which is to be protected, and store the checksum in a secure place. Whenever the file is used, the checksum must first be verified.



On the positive side, malware stored within a system cannot be completely invisible, since the malware code must be stored somewhere. This means that, in principle at least, malware can be detected by searching for characteristic sequences of information, called *signatures*.

Malware can, however, be (partially) hidden, either by splitting up the code, or having the malware store itself in encrypted form (*polymorphic* viruses) using a varying encryption key, which must also be stored with the encrypted code. However, the part of the malware code responsible for performing the decryption process must be left unencrypted, so that even polymorphic malware will have a (small) signature.

One fundamental limit of the signature approach to malware detection is that the malware must be known about, and hence it does not work against 'new' malware.

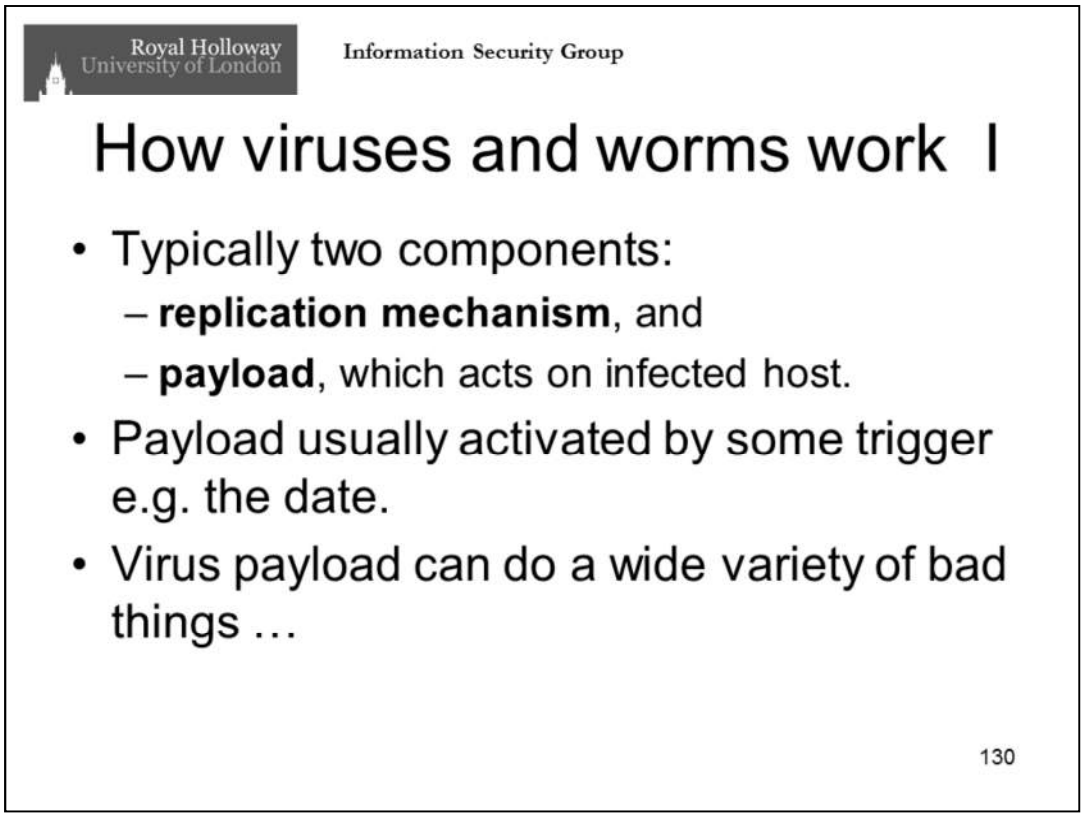


Virus/malware scanners search files for malware signatures, which can be used to identify infections by known strains of malware. Of course, the scanner will need to know about a type of malware to detect it, and hence such scanners need constant updating. Scanners are widely used, but they need to be efficient. They can significantly impair system performance if they are running constantly, which is the ideal way of using them.

Also, the threat remains of polymorphic malware, which can avoid detection since they will not have a single characteristic signatures. Also, *mutation engines* automate the generation of new malware strains, which can result in a constant supply of new malware whose signature may not be recognised by a scanner.

At one time signature-based scanners were regarded as a key defence, but in recent years their importance has declined.

We next examine viruses in greater detail.

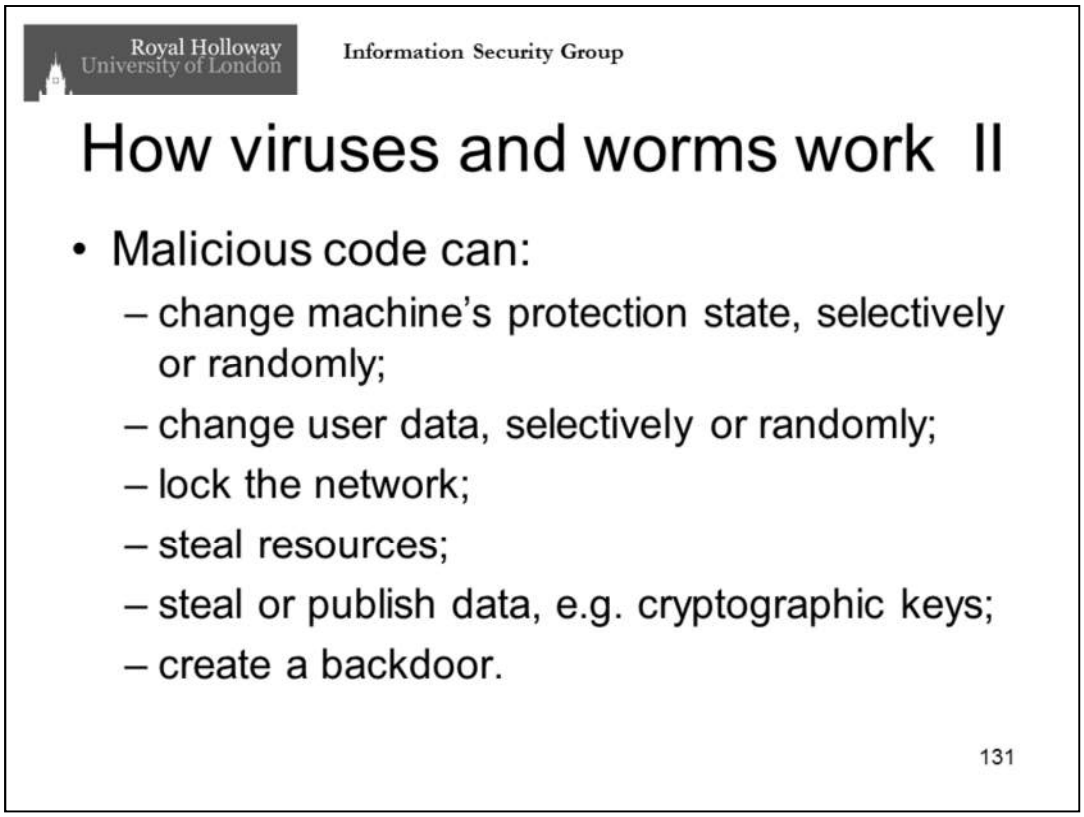


Typically, there are two components to a virus or worm:

• a replication mechanism, which enables the virus to make copies of itself, and

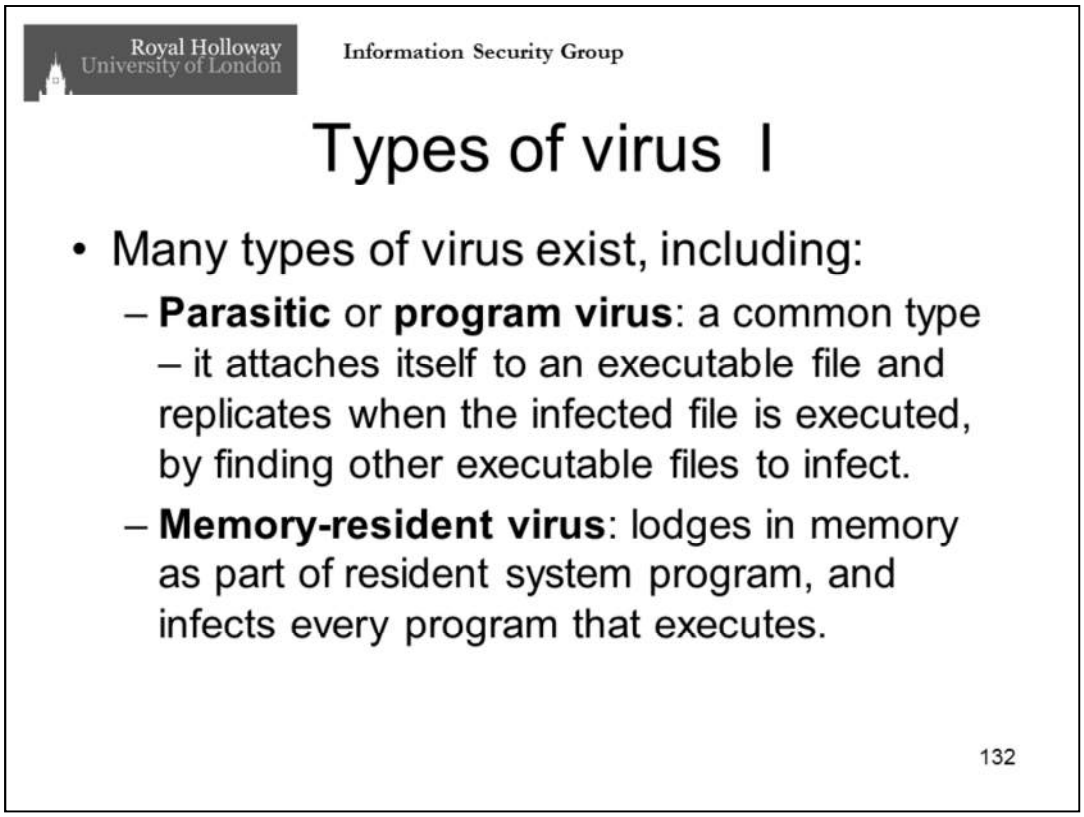
• a payload (which does something on the infected system to the advantage of the virus writer).

The payload is usually activated by some trigger. e.g. the date. A virus payload can do a wide variety of bad things, including installing additional malware, creating backdoors, etc. (see next slide).



The payload may, when triggered, do a variety of bad things:

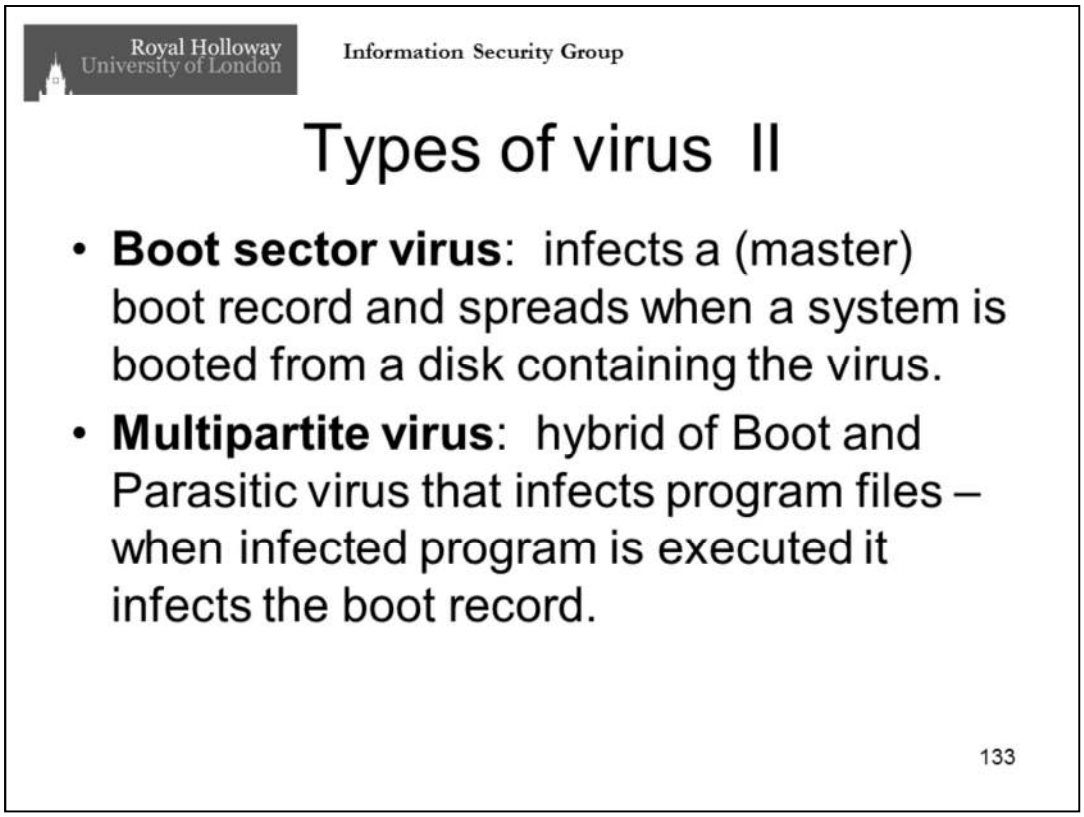
- make changes to the machine's protection state, selectively or randomly;
- make changes to user data, selectively or randomly, e.g. trash the disk;
- lock the network, e.g. by replicating at maximum speed;
- steal resources, e.g. use the CPU for a DES key search;
- steal, or publish your data like cryptographic keys – in Sept 2000 a virus was reported that steals the passwords used to access the home banking system of a Swiss bank;
- create a backdoor, so that the creator can take over the system later, perhaps to launch a distributed denial of service attack.



Virus researchers have put considerable effort into developing schemes for classifying computer viruses. Many types of virus can be identified, and among the most significant are the following (note that these categories are not mutually exclusive):

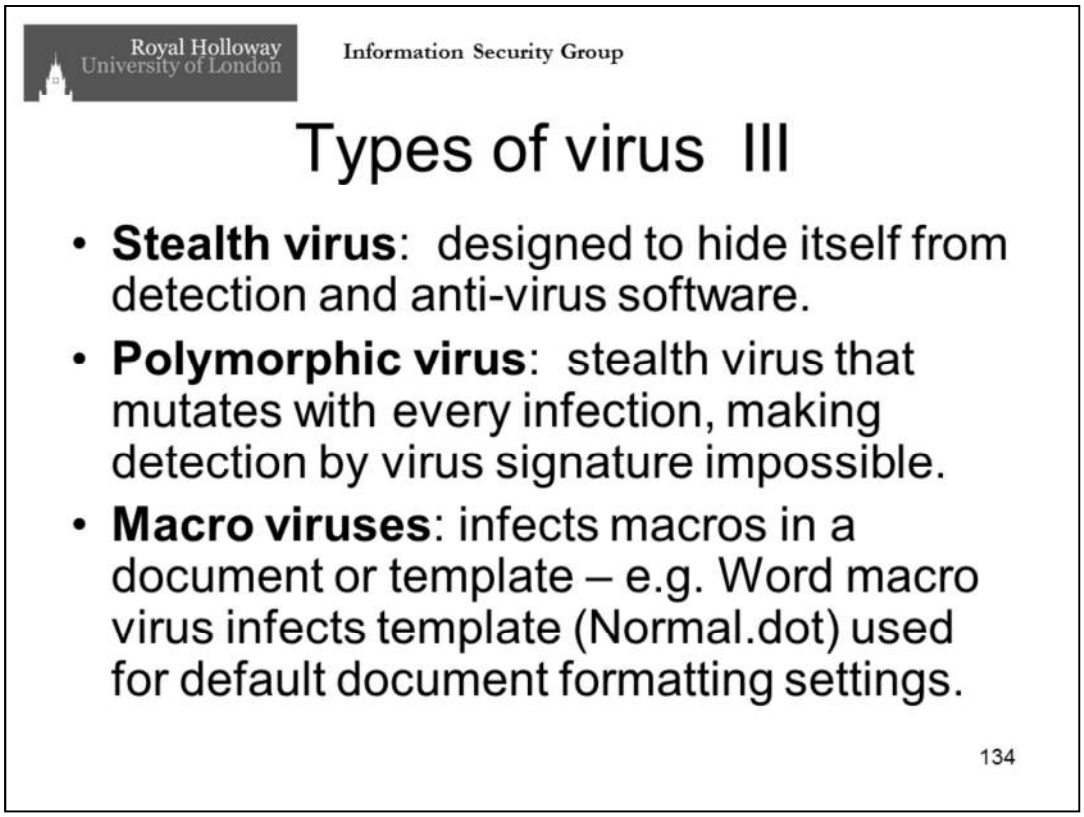
• *Parasitic* or *program virus*: a common form; such a virus attaches itself to executables typically by appending itself to the program, and inserting (at beginning of program) a jump to the viral code. At the end of the viral code there is a jump back into the program. It replicates when the infected file is executed, by finding other executable files to infect. That is, then the programs are loaded in memory during execution, the virus becomes active, making copies of itself and infecting files on disk. Examples include: Sunday, Cascade.

• *Memory-resident virus*: lodges in memory as part of a resident system program, and then infects every program that executes.



• *Boot sector virus*: infects a master boot record, or boot record and spreads when a system is booted from the disk containing the virus. Examples include: Form, Disk Killer, Michelangelo, and Stone virus.

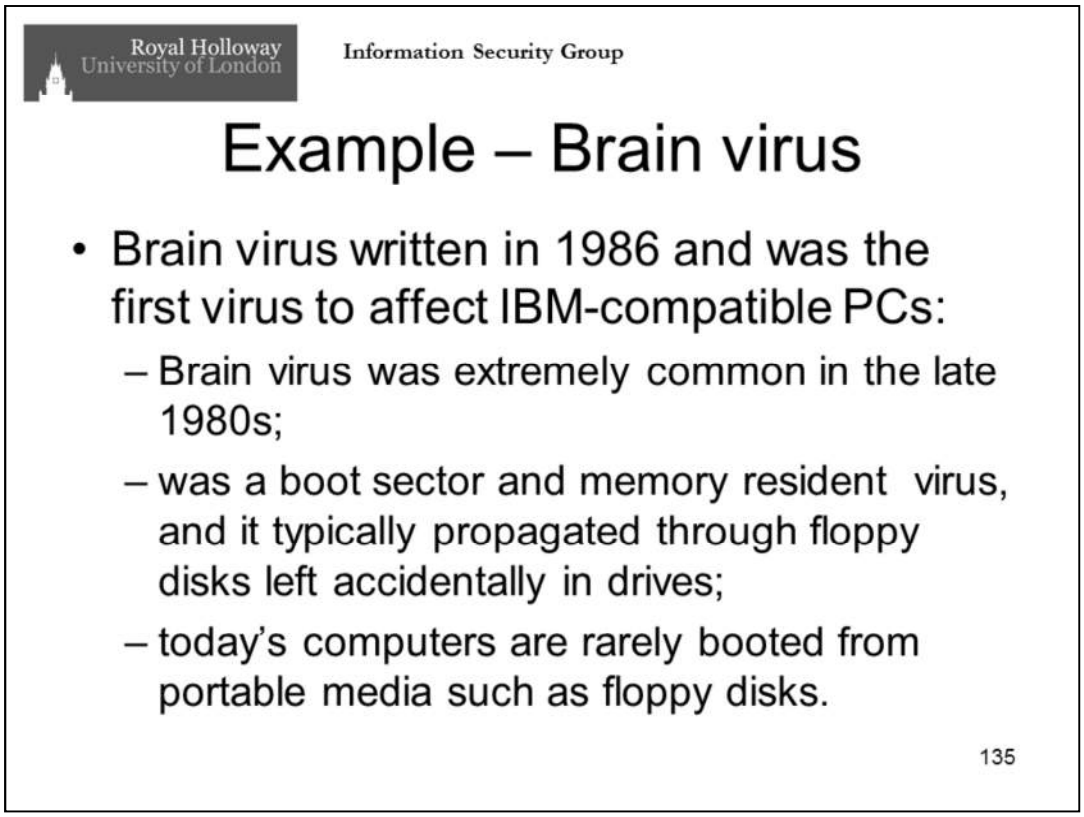
• *Multipartite viruses*: are a hybrid of Boot and Parasitic viruses. They infect program files, and when the infected program is executed these viruses infect the boot record. When you boot the computer next time the virus from the boot record loads in memory and then starts infecting other program files on disk. Examples include: Invader, Flip, and Tequila.



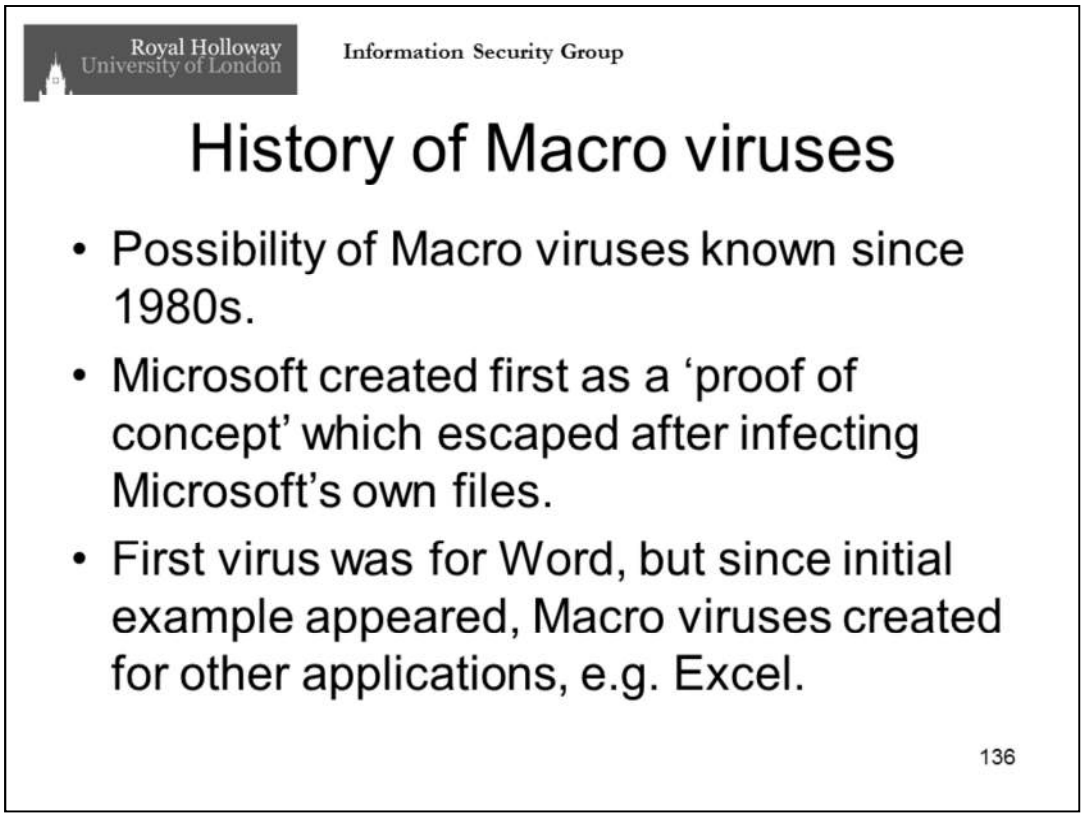
• *Stealth virus*: a form of virus explicitly designed to hide itself from detection and anti-virus software, e.g. it may use compression so that the infected program is exactly the same length as an uninfected version. Examples include: Frodo, Joshi, Whale.

• *Polymorphic virus*: a type of stealth virus that mutates with every infection, making detection by the signature of the virus (see later slide) difficult or impossible, i.e. it creates copies during replication that are functionally equivalent, but have different bit patterns. Examples include: Involuntary, Stimulate, Cascade, Phoenix, Evil, Proud, Virus 101.

• *Macro Viruses*: infects the macros within a document or template. For example, a Word macro virus is activated when you open a word processing or spreadsheet document, and it infects the Normal template (Normal.dot) - a general purpose file that stores default document formatting settings. Every document you open refers to the Normal template, and hence gets infected with the macro virus. Since this virus attaches itself to documents, the infection can spread if such documents are opened on other computers. Examples include: DMV, Nuclear, Word Concept.

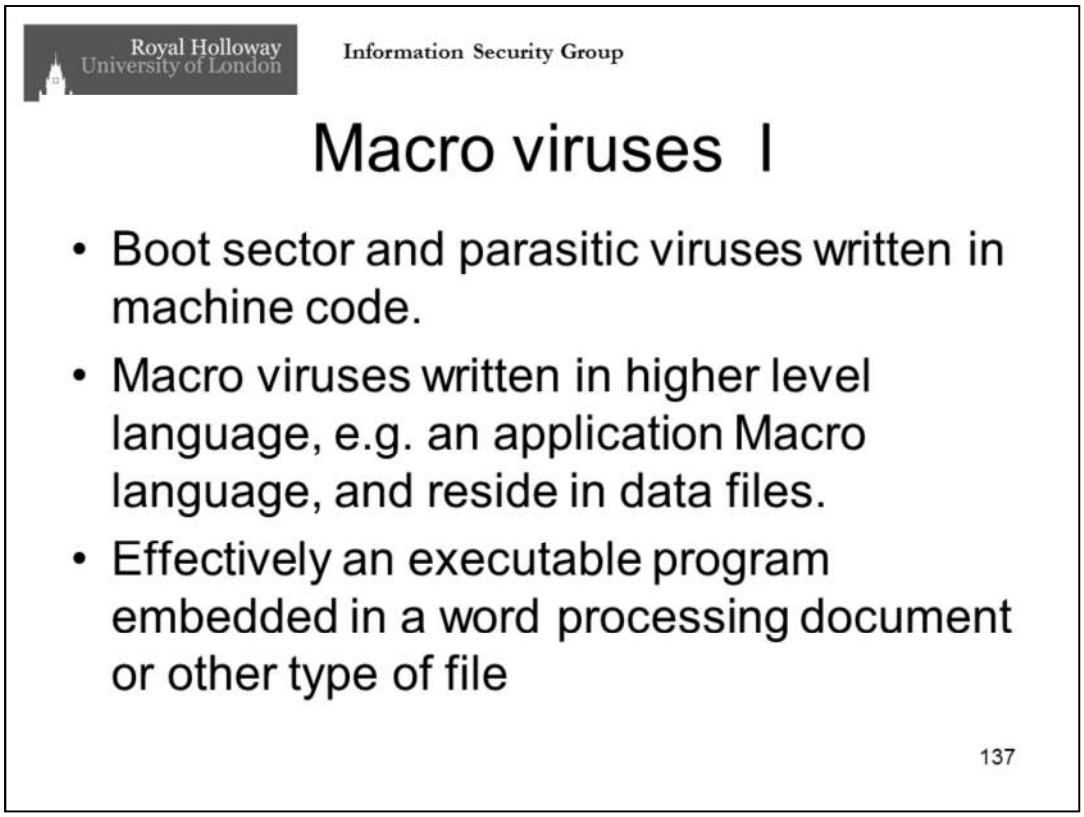


The Brain virus was written in 1986 and was the first virus to affect IBM-compatible PCs. It was a boot sector and memory resident virus, and it typically propagated through (infected) floppy disks left accidentally in drives – once memory resident it infected all floppy disks inserted in the machine. The Brain virus was extremely common in the late 1980s. Today computers are rarely booted from portable media such as floppy disks.



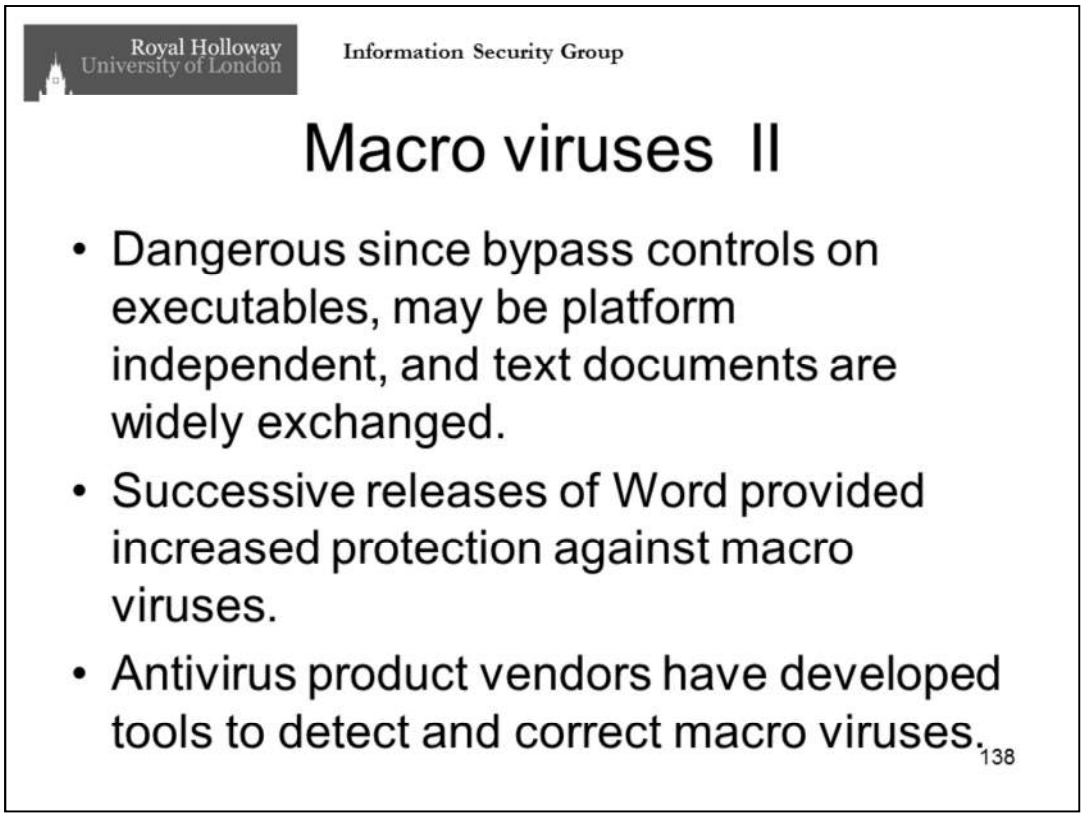
The possibility of Macro viruses has been known since at least the late 1980s. However, actual examples did not appear until well into the 1990s. In fact, Microsoft itself created the first known example of a Macro virus as a 'proof of concept' (the *Concept virus*) which escaped after infecting Microsoft's own files.

The first virus was for Microsoft Word, but since the initial example appeared, Macro viruses have been created for other applications, e.g. Excel.



The last type of viruses we consider are the so called *Macro viruses*. Whereas Boot sector and parasitic viruses are written in machine code, Macro viruses are written in a higher level language, e.g. an application Macro language, and reside in data files created by the relevant application.

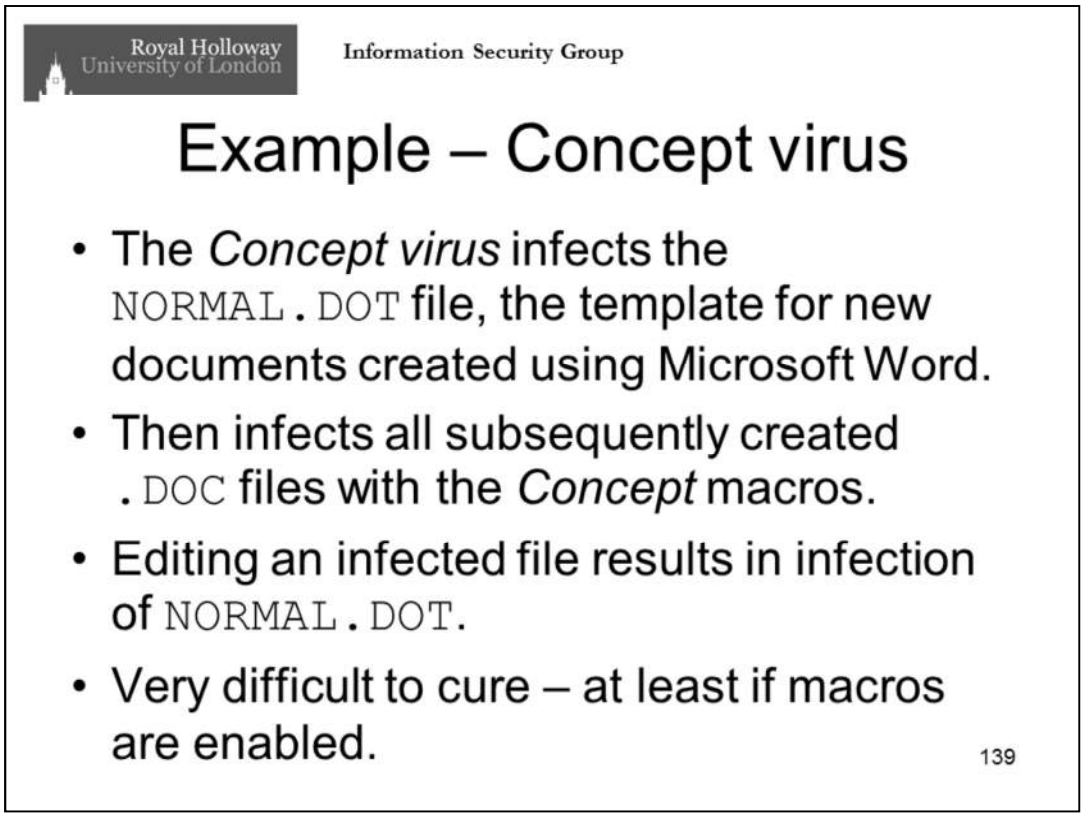
For example they can exist in word processing files, and would in that case be written in the Macro language for that word processor- usually some form of the Basic programming language and are often invoked automatically with no explicit user input. Once the macro is running, it can copy itself to other documents, delete files and cause other damage to the user's system.



Such viruses are very dangerous since they will bypass any controls on the distribution and use of executables, designed to counter parasitic viruses. Moreover, they may be platform independent. Most worrying of all, text documents are very widely exchanged in the commercial world, for legitimate business reasons (whereas the same is not generally true for executable files). Thus Macro viruses represent a serious long term corporate threat.

However, successive releases of Word have provided increased protection against macro viruses, and they are not the threat they once were.

Antivirus product vendors have developed tools to detect and correct macro viruses.

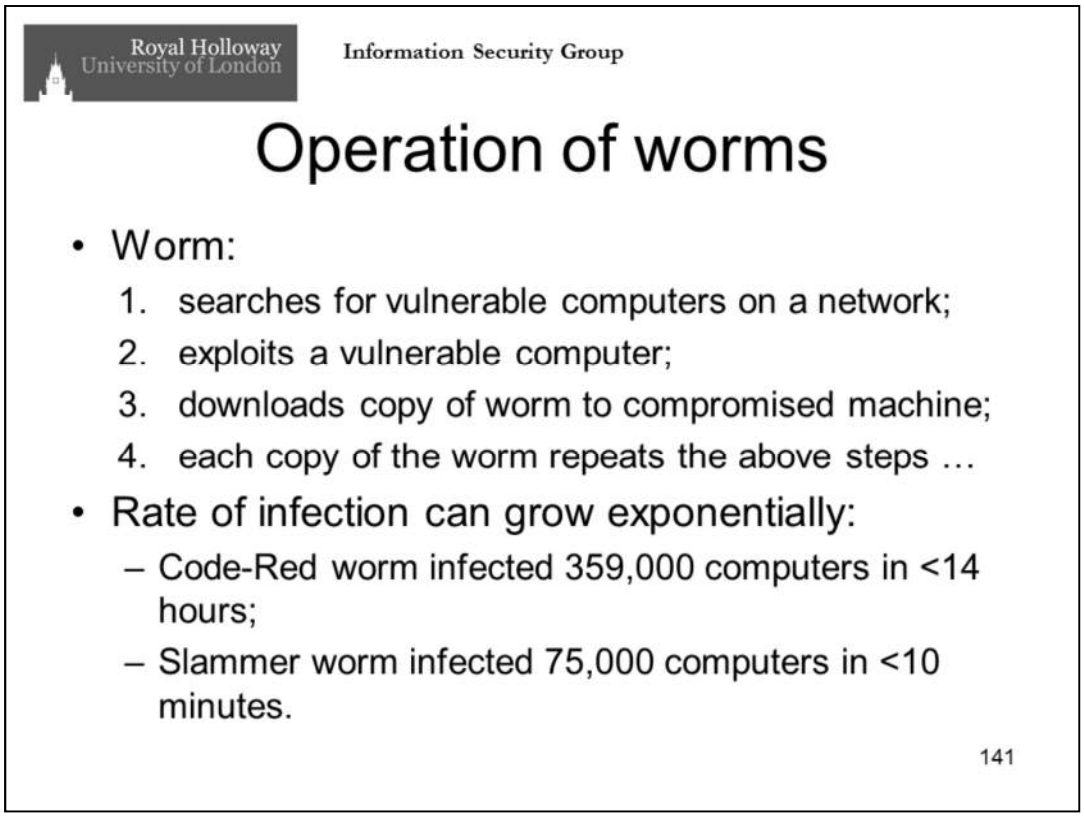


The *Concept virus* infects the `NORMAL.DOT` file, the template used when Microsoft Word creates a new document. Once loaded, the virus then infects all subsequently created `.DOC` files with the *Concept* macros.

If a running copy of Word is used to edit an infected file, then the virus macros are loaded into the running copy of Word, and when Word is shut down the `NORMAL.DOT` is infected.

Whilst it is possible to remove the virus from `NORMAL.DOT` (simply by deleting this file), it is very difficult to cure an infected file, and re-infection is very easy. All versions of Word from Word 97 onwards allow Macros to be disabled, preventing infections and gives a warning whenever a file containing macros is loaded.

We conclude this part of the course by discussing worms.



Put simply, a worm:

1. searches for vulnerable computers on a network, e.g. computers containing one or more specific software vulnerabilities;
2. exploits a vulnerable computer;
3. downloads a complete copy of the worm to the compromised computer;
4. each copy of the worm repeats the above steps …

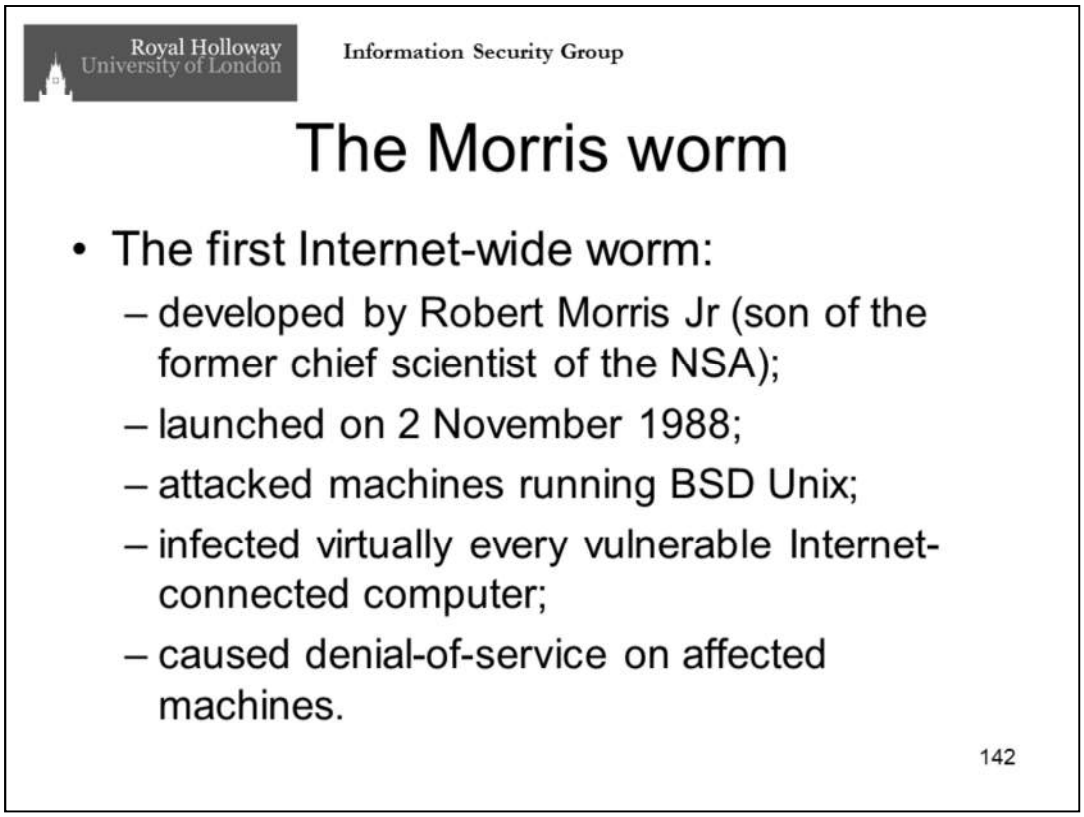
The rate of infection can grow exponentially:

• the Code-Red worm infected 359,000 computers in less than 14 hours – see:

http://www.caida.org/research/security/code-red/coderedv2_analysis.xml

• the Slammer worm infected over 75,000 computers in less than 10 minutes – see:

http://www.cs.ucsd.edu/~savage/papers/IEEESP03.pdf

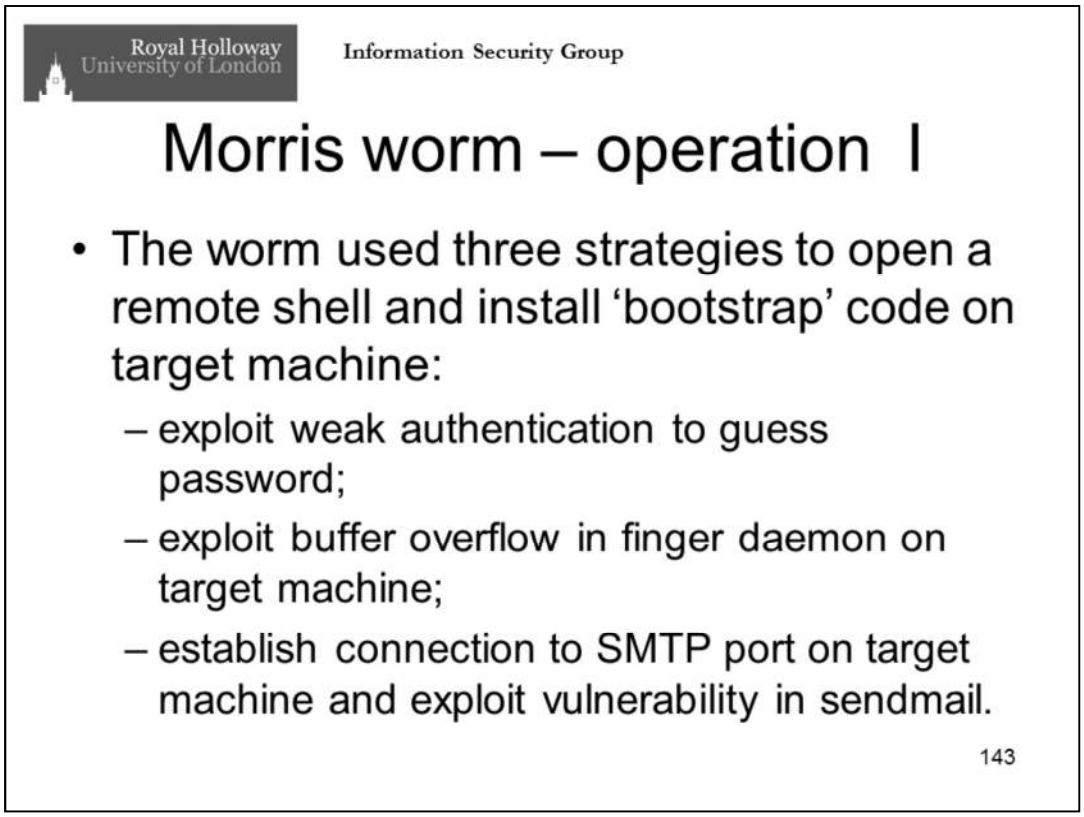


The Morris worm was the first Internet-wide worm. It:

- was developed by Robert Morris Jr (son of the former chief scientist of the NSA);
- was launched on 2 November 1988;
- attacked machines running BSD Unix;
- infected virtually every vulnerable Internet-connected computer;
- caused denial-of-service on affected machines.

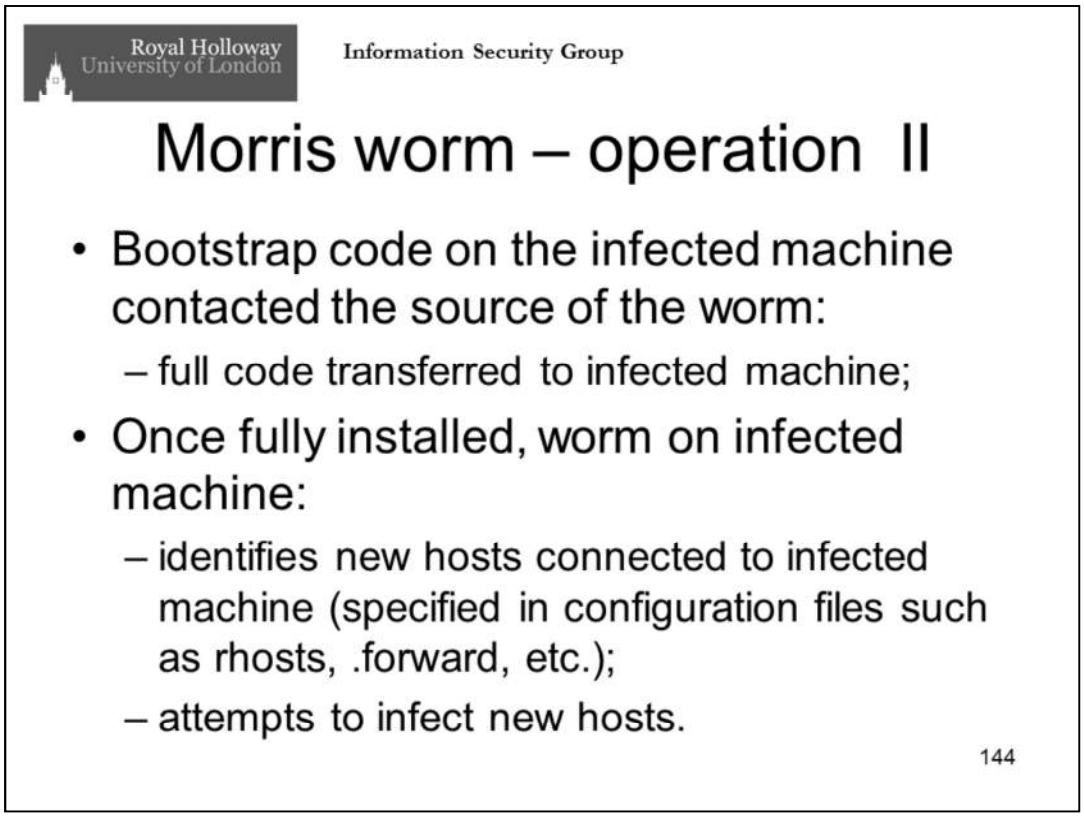
For further details see (for example):

http://homes.cerias.purdue.edu/~spaf/tech-reps/823.pdf



The worm used three strategies to open a remote shell and install special 'bootstrap' code on a target machine:

- exploit weak authentication to guess password;
- exploiting a buffer overflow in the finger daemon on the target machine;
- establish a connection to the SMTP port on the target machine and exploit a vulnerability in the sendmail program.

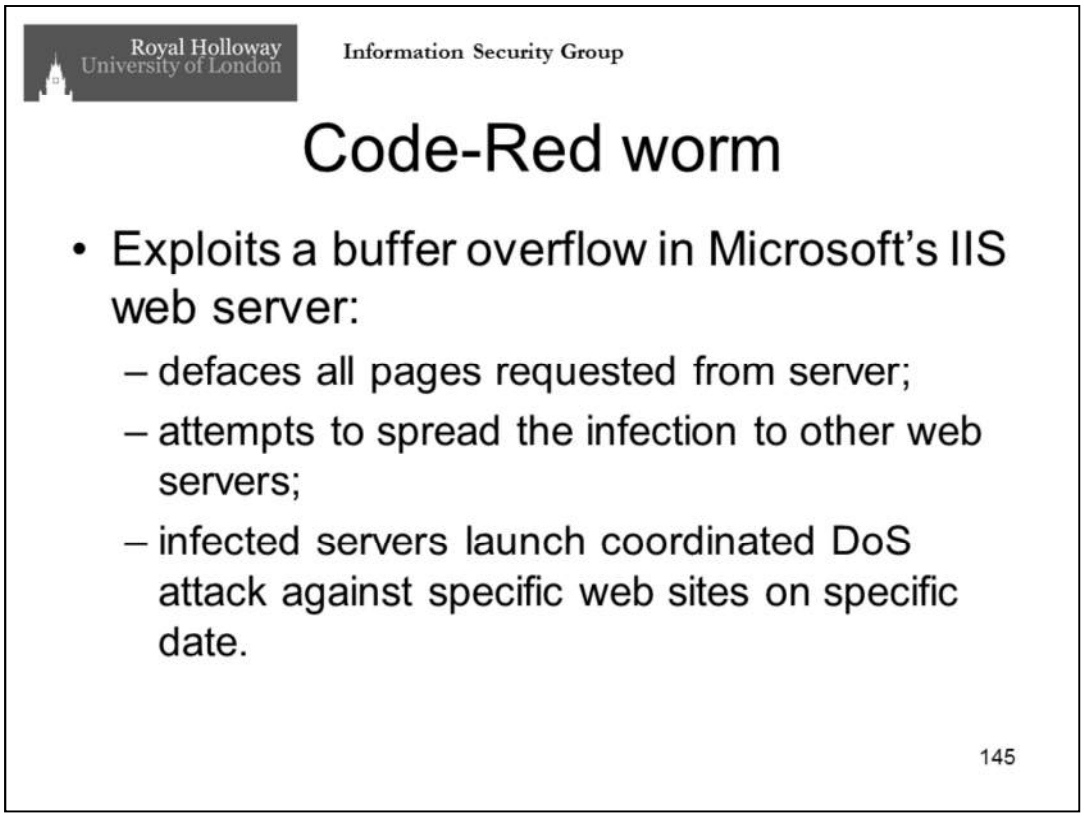


The bootstrap code on the infected machine contacted the source of the worm, and the full code of the worm was then transferred to the infected machine.

Now fully installed, the worm on the infected machine:

• identifies new hosts connected to the infected machine (specified in configuration files such as `rhosts`, `.forward`, etc.);

• attempts to infect new hosts.



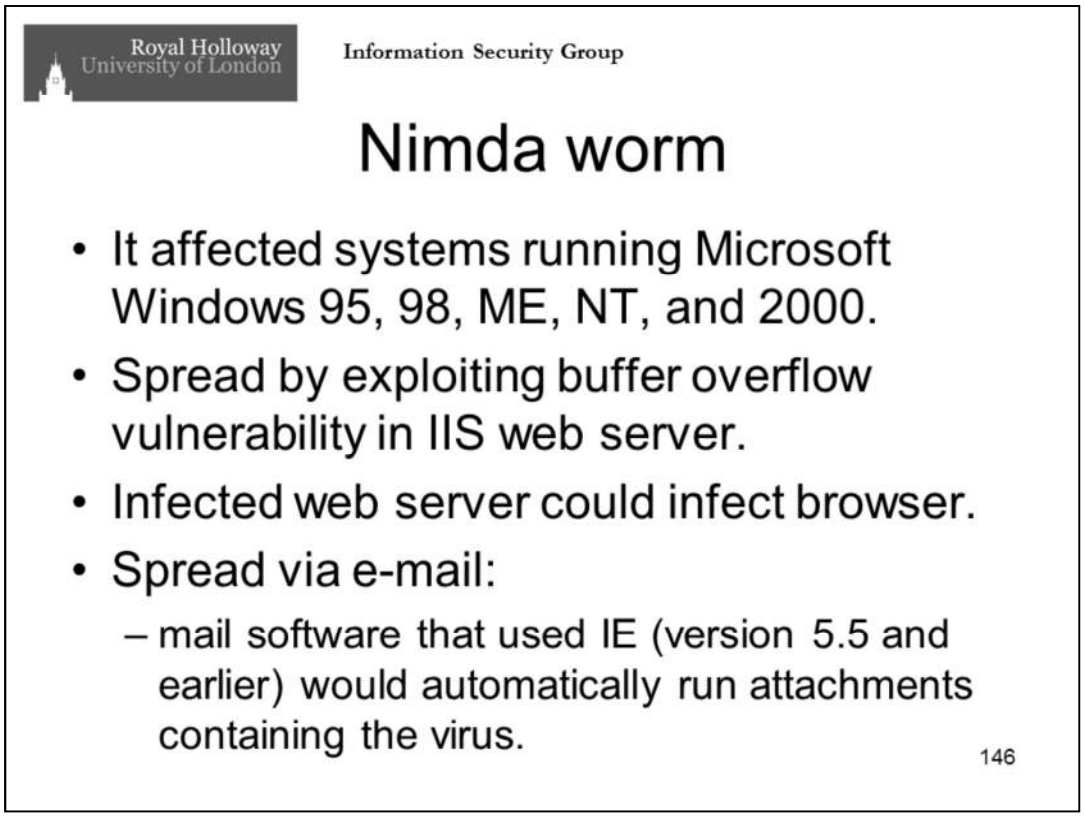
This worm exploits a buffer overflow in Microsoft's IIS web server. It:

- defaces all pages requested from server;
- attempts to spread the infection to other web servers.

Infected servers launch coordinated denial of service (DoS) attack against specific web sites on a specific date.

For further details see:

http://www.cert.org/advisories/CA-2001-19.html



The Nimda worm affected systems running Microsoft Windows 95, 98, ME, NT, and 2000.

It spread by exploiting buffer overflow vulnerability in IIS web server.

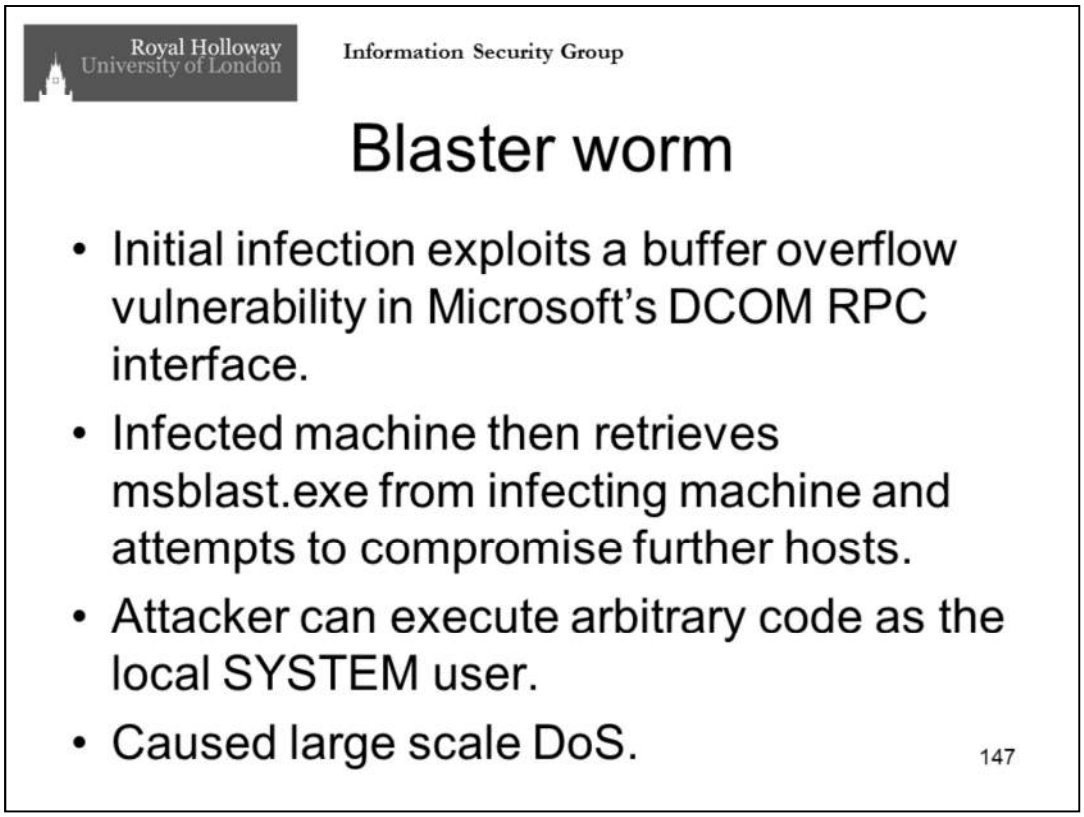
Infected web server could infect browser.

It spread via e-mail:

• mail software that used IE (version 5.5 and earlier) would automatically run attachments containing the virus.

For further details see:

http://www.cert.org/advisories/CA-2001-26.html



Initial infection exploits a buffer overflow vulnerability in Microsoft's DCOM RPC interface.

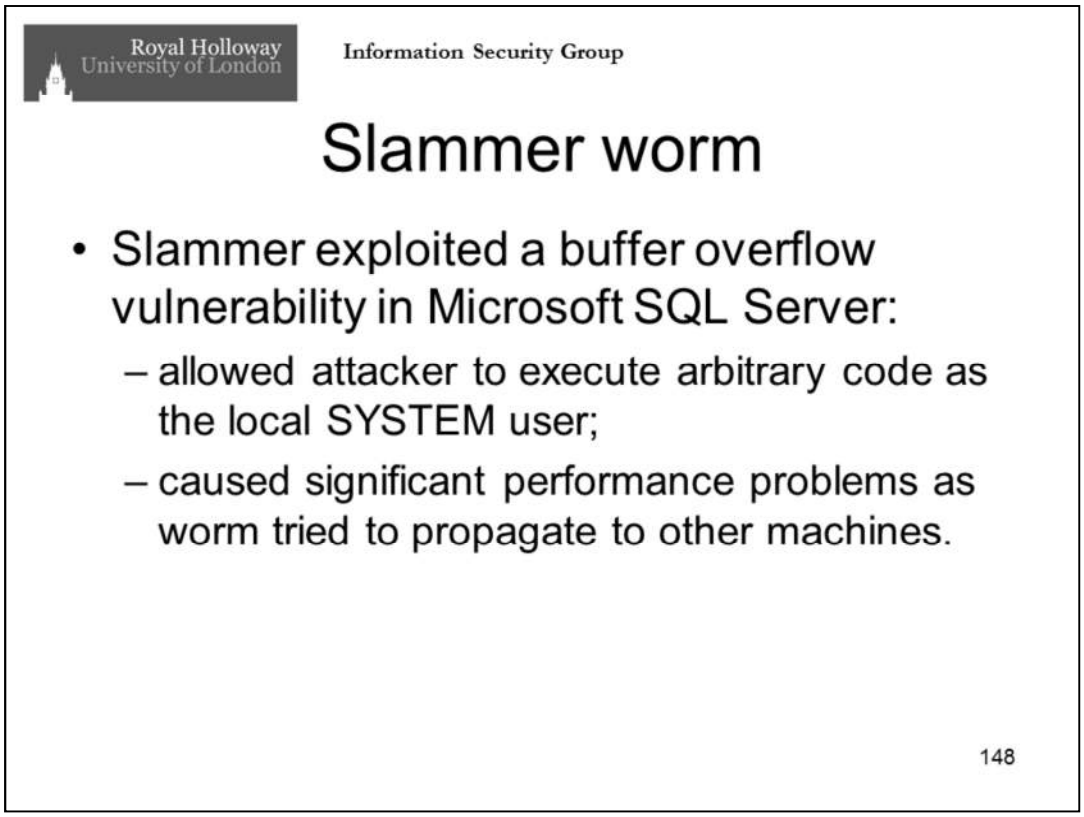
Infected machine then retrieves msblast.exe from the infecting machine and attempts to compromise further hosts.

Attacker can execute arbitrary code as the local SYSTEM user.

It caused large scale denial of service.

For further details see:

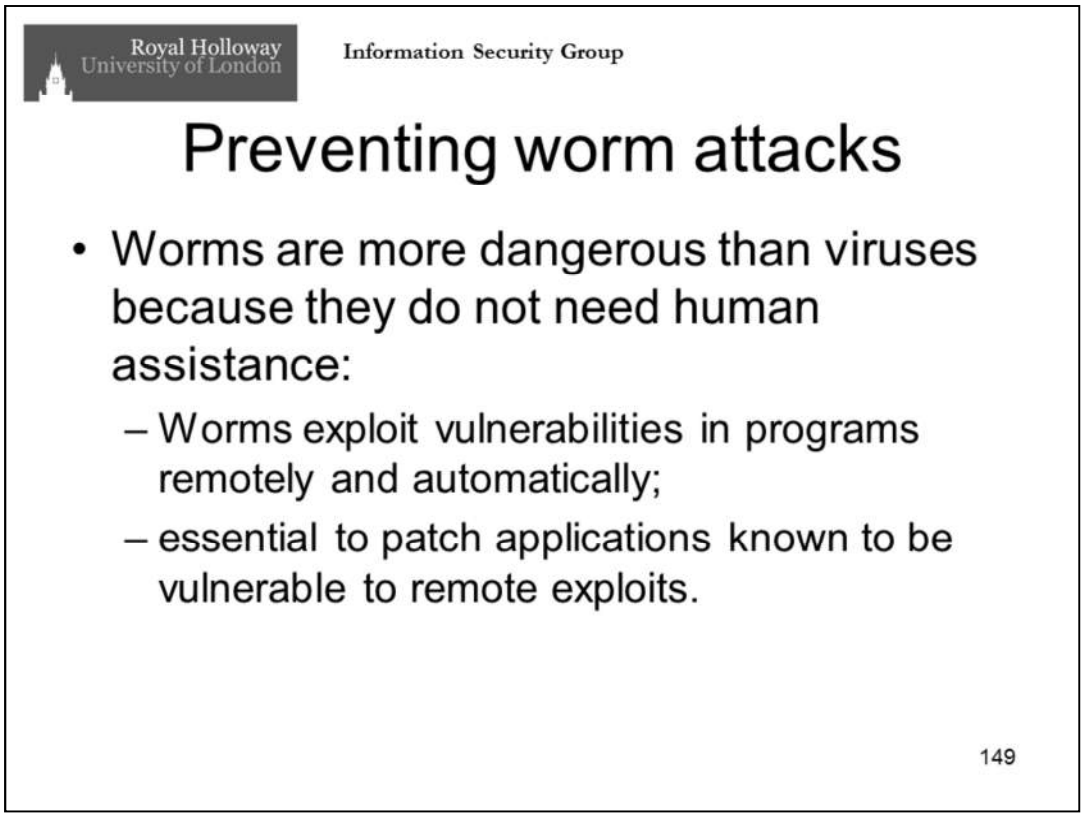
http://www.cert.org/advisories/CA-2003-20.html



The Slammer worm exploited a buffer overflow vulnerability in Microsoft SQL Server. It allowed the attacker to execute arbitrary code as the local SYSTEM user. It caused significant performance problems as the worm tried to propagate to other machines.

For further details see:

http://www.cert.org/advisories/CA-2003-04.html

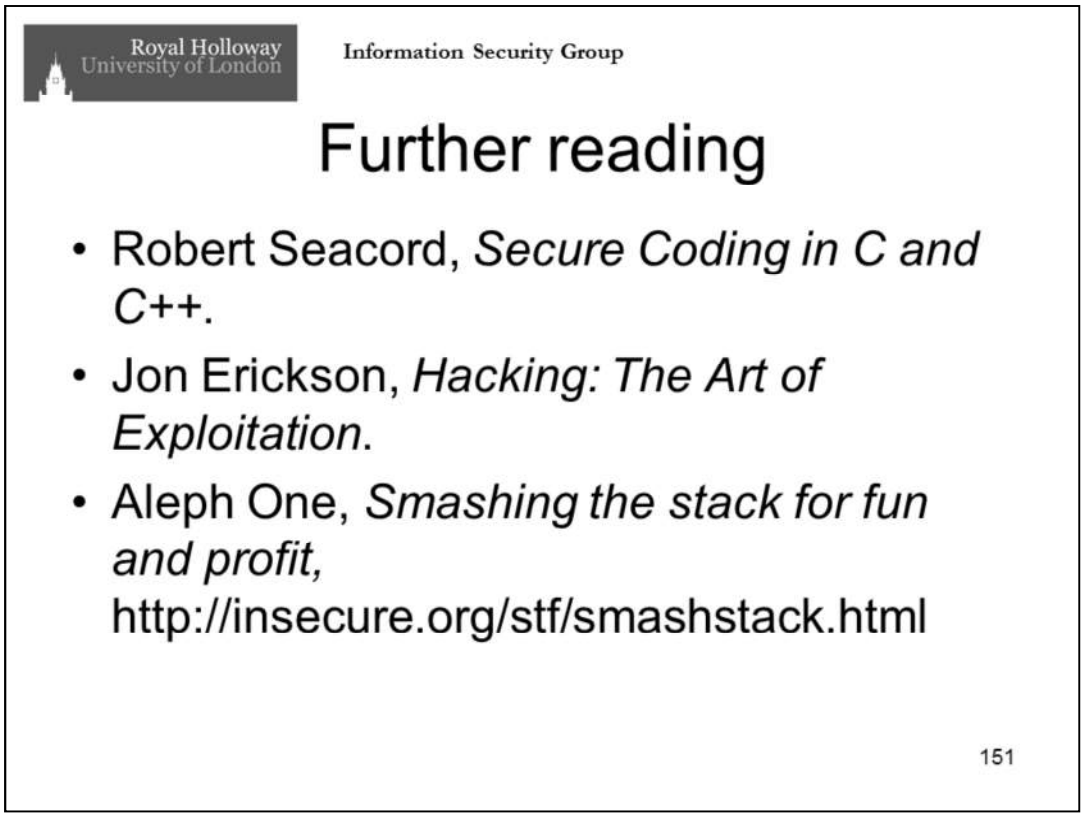


Worms are more dangerous than viruses because they do not need human assistance:

- Worms exploit vulnerabilities in programs remotely and automatically;
- it is thus essential to patch applications that are known to be vulnerable to remote exploits.

Because they work without human assistance, they can spread very fast, much faster than a typical virus infection.


Royal Holloway
University of London
Information Security Group
Agenda
• Threats to security
• Managing security vulnerabilities
• Software vulnerabilities
• Other vulnerabilities
• Malware
• Viruses
• Worms
• Resources
150



In addition to the links given in the body of the presentation, the following reading is recommended.

- Robert Seacord, *Secure Coding in C and C++*.
- Jon Erickson, *Hacking: The Art of Exploitation*.
- Aleph One, *Smashing the stack for fun and profit,* http://insecure.org/stf/smashstack.html

Key links given in the body of the course material include:

- the US-CERT home page (http://www.us-cert.gov/), which has a wide variety of useful publications;
- the Microsoft Security Intelligence Report (http://www.microsoft.com/sir) is published twice a year, and contains a huge amount of useful information.

Finally, there is a huge amount of information available on the Internet about specific instances of malware and computer system vulnerabilities.