Database security as compared to OS security

- Operating systems manage data
- Users create, read, write, and delete data
- These operations do not really consider content
- So neither does the OS access control decisions

- Databases manage information
- Users enter and extract information in the database
- The task of the database is to maintain and give access to the stored information
- So database access control must consider the content in access control decisions
Database content is complicated to protect

• To protect users checking who has a high salary, simply prohibit searches that lists (high) salaries, or?

• There are many ways to get at database information
  • Exact data queries
  • Upper or lower bounds
  • Existence of data
  • Negative results
  • Probable value (or statistics)

• Remember that you do want information to be available

• Even sensitive information should be available to users with the correct privileges

• Your protection needs to be very precise
Database content

- The information in the database relates to the outside world
- and often to other info in the database
- Therefore, the information should be kept
  - Internally consistent
  - Externally consistent
- The DataBase Management System (DBMS) lives in the services layer
- Sometimes it is also used to define security controls in the application layer
Subjects and authentication in databases

- A database can accept operating system authentication
- Or perform user authentication on its own
- Advantages and drawbacks for either strategy are exactly as for general distributed systems
  - OS authentication must pass the result in a secure and trusted way to the DBMS
  - Internal authentication reduces TOCTTOU problem, because reauthentication is simpler
  - It also enables more simple handling of several user identities/privilege levels
Integration with the OS
Example of database structure

• A relational database is perceived by its users as a collection of tables
• (But the internal structure is often different)

<table>
<thead>
<tr>
<th>Name</th>
<th>Day</th>
<th>Flight</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Mon</td>
<td>GR123</td>
<td>private</td>
</tr>
<tr>
<td>Bob</td>
<td>Mon</td>
<td>YL011</td>
<td>business</td>
</tr>
<tr>
<td>Bob</td>
<td>Wed</td>
<td>BX201</td>
<td></td>
</tr>
<tr>
<td>Carol</td>
<td>Tue</td>
<td>BX201</td>
<td>business</td>
</tr>
<tr>
<td>Alice</td>
<td>Thu</td>
<td>FL9700</td>
<td>business</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flight</th>
<th>Dest</th>
<th>Departure</th>
<th>Days</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR123</td>
<td>LPI</td>
<td>7:55</td>
<td>1-4-</td>
</tr>
<tr>
<td>YL011</td>
<td>ATL</td>
<td>8:10</td>
<td>12345-7</td>
</tr>
<tr>
<td>BX201</td>
<td>SLA</td>
<td>9:20</td>
<td>1-3-5-</td>
</tr>
<tr>
<td>FL9700</td>
<td>SLA</td>
<td>14:00</td>
<td>-2-4-6-</td>
</tr>
<tr>
<td>GR127</td>
<td>LPI</td>
<td>14:55</td>
<td>-2-5-</td>
</tr>
</tbody>
</table>
Databases, operations

- The standard operation language for relational databases is Structured Query Language, SQL
- Data access is through
  - SELECT
  - UPDATE
  - INSERT
  - DELETE
- Privileges are managed with
  - GRANT
  - REVOKE
Databases, operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Day</th>
<th>Flight</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Mon</td>
<td>GR123</td>
<td>private</td>
</tr>
<tr>
<td>Bob</td>
<td>Mon</td>
<td>YL011</td>
<td>business</td>
</tr>
<tr>
<td>Bob</td>
<td>Wed</td>
<td>BX201</td>
<td></td>
</tr>
<tr>
<td>Carol</td>
<td>Tue</td>
<td>BX201</td>
<td>business</td>
</tr>
<tr>
<td>Alice</td>
<td>Thu</td>
<td>FL9700</td>
<td>business</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flight</th>
<th>Dest</th>
<th>Departure</th>
<th>Days</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR123</td>
<td>LPI</td>
<td>7:55</td>
<td>1-4-</td>
</tr>
<tr>
<td>YL011</td>
<td>ATL</td>
<td>8:10</td>
<td>12345-7</td>
</tr>
<tr>
<td>BX201</td>
<td>SLA</td>
<td>9:20</td>
<td>1-3-5-</td>
</tr>
<tr>
<td>FL9700</td>
<td>SLA</td>
<td>14:00</td>
<td>-2-4-6-</td>
</tr>
<tr>
<td>GR127</td>
<td>LPI</td>
<td>14:55</td>
<td>-2-5-</td>
</tr>
</tbody>
</table>

SELECT Name FROM Diary
WHERE Flight IN

( SELECT Flight FROM Flights
  WHERE Destination = 'LPI' )
Relational database objects

• It is easy to think that the objects are the “tables where data is stored”
• But the information is really in the relations
• These come in four flavours
  • *Base relations, or real relations*, named autonomous relations that have associated stored data
  • *Views*, named derived relations with no stored data
  • *Snapshots*, named derived relations with stored data
  • *Query results*, may have a name but have no persistent existence
• Note that part of a view, a *column* of attributes can also be a security-relevant object
Database keys

- Each entry in a relation should have an identity of its own, a *key*.
- Sometimes a single attribute is enough, sometimes the key needs to be composite.
- A *primary key* is a unique and minimal identifier for that relation.
- When the primary key is an attribute in another relation, it is known as a *foreign key* in that relation.
- There are two internal consistency rules that apply to these:
  - No component of a primary key in a base relation should be empty.
  - No relation may contain an unmatched foreign key value.
The SQL security model

- Security in any DBMS should be
  - **Complete**, protect all fields in the database
  - **Consistent**, so that no conflicting rules exist

- SQL implements discretionary access control for the triple
  
  \[(\text{user}, \text{action}, \text{object})\]

- An owner is assigned at creation

- The owner can then grant access through giving a *privilege* that consists of
  
  \[(\text{grantor}, \text{grantee}, \text{object}, \text{action}, \text{grantable})\]
Granting privileges

- Granting access is through the GRANT operation
  
  ```sql
  GRANT SELECT, UPDATE (Day, Flight)
  ON TABLE Diary
  TO TravelAgent1, TravelAgent2
  ```

- Revoking is through REVOKE
  
  ```sql
  REVOKE UPDATE
  ON TABLE Diary
  FROM TravelAgent2
  ```
Granting granting privileges

• There is a GRANT option that can be added

  GRANT SELECT
  ON TABLE Diary
  TO TravelAgent1
  WITH GRANT OPTION

• Revoking is through REVOKE

  REVOKE SELECT
  ON TABLE Diary
  FROM TravelAgent1

• But now, all privileges granted by TravelAgent1 also need to be revoked (by the database system)
Views as a security tool

- You could implement access control in the base relations, but this will be complicated and error-prone.
- It is better to use Views, derived (but lasting) relations that can be thought of as selected parts of tables.

```
CREATE VIEW business_trips AS
  SELECT * FROM Diary
  WHERE Status = 'business'
```

- By creating views, you get fine-tuned access control, say give access to one field depends on the value in another.
- Access control through views can be seen as being in the application layer, and the DBMS is the service that implements it.
View examples

• Views can even be defined relative to the current user

    CREATE VIEW My_Journeys AS
    SELECT * FROM Diary
    WHERE Customer = current_user()

• More advanced views can compare attributes

    CREATE VIEW Top_of_the_class (Name, Grade) AS
    SELECT * FROM Students WHERE Grade <
    (SELECT Grade FROM Students
     WHERE Name = current_user() )

• This can give very fine-grained access control

• It is easy to implement group access control, and user’s right to grant and revoke access control
Using views to UPDATE

- There are views that simply cannot be used to UPDATE, for example when the needed primary key is not in the view
- Another problem is when writes make items disappear

```
CREATE VIEW business_trips AS
    SELECT * FROM Diary
    WHERE Status = 'business'
```

- If a travel agent has UPDATE access to the view, she may want to make Alices thursday trip(s) confidential

  ```
  UPDATE business_trips
  SET Status = 'private'
  WHERE Name='Alice' AND Day = 'Thu'
  ```

- The item drops from view, maybe even for the travel agent herself
- This may not be appropriate, prevented by WITH CHECK OPTION
Using views to UPDATE

• There are views that simply cannot be used to UPDATE, for example when the needed primary key is not in the view

• Another problem is when writes make items disappear

CREATE VIEW business_trips AS
SELECT * FROM Diary
WHERE Status = 'business'
WITH CHECK OPTION

• If a travel agent has UPDATE access to the view, she may want to make Alices thursday trip(s) confidential

UPDATE business_trips
SET Status = 'private'
WHERE Name='Alice' AND Day = 'Thu'

• The item drops from view, maybe even for the travel agent herself

• This may not be appropriate, prevented by WITH CHECK OPTION
Access control with views

- There are benefits
  - Views are flexible and policies can be described close to application needs
  - Views can be context- and data-dependent
  - Views can be seen as controlled invocation
  - Secure views can replace security labels
  - Data can be easily reclassified

- and disadvantages
  - Checking can become complicated and slow
  - Views need to be checked: do they give the desired security?
  - Completeness and consistency are not achieved automatically
  - Security part of DBMS may become large
  - Views are less suitable when focus is security of individual data items (rather than restricting user actions)

- More can be said about SQL access control, and some DBMS have more expressive access control interfaces too
Indirect inference of confidential values

- In a statistical database, individual data may be protected
- Statistical data can be used to infer protected values
- SQL have the following aggregate functions
  - COUNT
  - SUM
  - AVG
  - MAX
  - MIN
- In this setting, there will be information flow to the user
- Our task is now to keep the flow at an acceptable level
### Example

<table>
<thead>
<tr>
<th>Name</th>
<th>Sex</th>
<th>Prog</th>
<th>Units</th>
<th>Grade avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>F</td>
<td>MBA</td>
<td>8</td>
<td>63</td>
</tr>
<tr>
<td>Bill</td>
<td>M</td>
<td>CS</td>
<td>15</td>
<td>58</td>
</tr>
<tr>
<td>Carol</td>
<td>F</td>
<td>CS</td>
<td>16</td>
<td>70</td>
</tr>
<tr>
<td>Don</td>
<td>M</td>
<td>MIS</td>
<td>22</td>
<td>75</td>
</tr>
<tr>
<td>Errol</td>
<td>M</td>
<td>CS</td>
<td>8</td>
<td>66</td>
</tr>
<tr>
<td>Flora</td>
<td>F</td>
<td>MIS</td>
<td>16</td>
<td>81</td>
</tr>
<tr>
<td>Gala</td>
<td>F</td>
<td>MBA</td>
<td>23</td>
<td>68</td>
</tr>
<tr>
<td>Homer</td>
<td>M</td>
<td>CS</td>
<td>7</td>
<td>50</td>
</tr>
<tr>
<td>Igor</td>
<td>M</td>
<td>MIS</td>
<td>21</td>
<td>70</td>
</tr>
</tbody>
</table>

```
SELECT AVG(Grade avg)
FROM Students
WHERE Programme = 'MBA'
```
Aggregation and inference

- **Aggregation** refers to the observation that the security level of an aggregate may differ from that of individual elements.

- The **inference problem** is that sensitive information can be derived from insensitive data.
  - A *direct attack* is using a small sample so that information leaks directly.
  - An *indirect attack* combining several aggregates to infer information.

- A *tracker attack* is a particular type of indirect attack which is present in most statistical databases.

- A *linear system vulnerability* is an algebraic relation between query responses.
Trackers

• An individual tracker is a predicate, which uniquely points out the target, when used in an aggregate question

```
SELECT AVG(Grade avg)
FROM Students
WHERE Sex = 'F' AND Programme = 'MBA'
```

• With the individual tracker as the predicate in a question, you can get the value of unknown attributes for a target, without being authorized to directly select them

```
SELECT COUNT(*)
FROM Students
WHERE Sex = 'F' AND Programme = 'MBA'
```

• Individual trackers can be prevented by not allowing questions that point out only one (or all but one) entry in the database
General trackers

- Unfortunately, this is not enough
- Even if each statistical query must return several answers, individual data may still be accessible
- A *general tracker* is a predicate that can be used to find the answer to any inadmissible query
- Basically, it is a query for which both the query set and its complement are large enough to allow the query
- Almost all statistical databases have a general tracker
General tracker, example

<table>
<thead>
<tr>
<th>Name</th>
<th>Sex</th>
<th>Prog</th>
<th>Units</th>
<th>Grade avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>F</td>
<td>MBA</td>
<td>8</td>
<td>63</td>
</tr>
<tr>
<td>Bill</td>
<td>M</td>
<td>CS</td>
<td>15</td>
<td>58</td>
</tr>
<tr>
<td>Carol</td>
<td>F</td>
<td>CS</td>
<td>16</td>
<td>70</td>
</tr>
<tr>
<td>Don</td>
<td>M</td>
<td>MIS</td>
<td>22</td>
<td>75</td>
</tr>
<tr>
<td>Errol</td>
<td>M</td>
<td>CS</td>
<td>8</td>
<td>66</td>
</tr>
<tr>
<td>Flora</td>
<td>F</td>
<td>MIS</td>
<td>16</td>
<td>81</td>
</tr>
<tr>
<td>Gala</td>
<td>F</td>
<td>MBA</td>
<td>23</td>
<td>68</td>
</tr>
<tr>
<td>Homer</td>
<td>M</td>
<td>CS</td>
<td>7</td>
<td>50</td>
</tr>
<tr>
<td>Igor</td>
<td>M</td>
<td>MIS</td>
<td>21</td>
<td>70</td>
</tr>
</tbody>
</table>

SELECT SUM(Units) FROM Students
WHERE Name = 'Carol' OR Programme = 'MIS'

SELECT SUM(Units) FROM Students
WHERE Name = 'Carol' OR NOT (Programme = 'MIS')

SELECT SUM(Units) FROM Students
Further inference 1

- Portia has taken an exam. Has she flunked again?
  - How many didn’t pass? If the answer is 0, she made it this time
- Portia has passed, what can her grade be?
  - You and your best friend got grade 3. If the number of students with grade 3 is 2, Portia did not get 3.
  - The maximum grade given at this retake was 4. So if Portia passed and did not have grade 3, she obviously has 4.
Some cars in a fleet have a sensitive cargo. Their identity and route must be kept from all except specially cleared employees. Another employee tries to get their values.

- Ask for the number of cars in the fleet.
- Ask for a list of available cars
- Get a list of the cargo of all cars. This must be available for any administrator for normal cargoes. Sensitive cargoes will not be on the list.
- Are some cars missing from both lists? They have the sensitive cargo...
Further inference 3

• An administrator can do a lot of statistics on the department salaries in order to keep track of the department economy, but cannot see individual salaries. What is the salary of the department head?

• Exact techniques, if allowed
  • What is the max salary for the department?
  • How many individual salaries are higher than $x$? Modify $x$...

• Indications
  • How many department salaries are between $x$ and $y$, where $x$ and $y$ must, to prevent direct inference, be from a preset set of allowed values and $y - x > z$ so that no exact answer can be found.
Tracker defence and defence limits

- Suppress obviously sensitive information, check query set size
- Disguise the data
  - Randomly swap entries
  - Add random perturbation to the data
- Protection of this kind reduces usability of the data
- Analyze the database to find sensitive attributes, and put these in a separate table; medical statistical databases are often anonymized
- Track the queries for each user, and keep a log of what each user knows
  - This gives good protection but is also expensive
  - Especially when users may be cooperating
Software security

• Security and reliability are both about unexpected problems
• Reliability is about purely accidental failures
• The probability that a failure will happen follows a certain distribution
• It does not matter how many bugs there are, it matters how often they are triggered
• Testing is against expected usage
• Attackers do the unexpected, and effectively chooses the distribution
• You’d want your code bug-free
• Experience shows that the number of bugs decrease exponentially
Malware taxonomy

- *Malware* is any software with a malicious purpose
- *Computer viruses* are self-replicating code pieces that *infect* other legitimate programs and files
- *Worms* are self-replicating code pieces that spread on their own
- *Trojan horses* are programs with a legitimate purpose but also with hidden malicious functions
Hackers

• *Hackers* initially referred to people with intimate knowledge about programming and computer systems

• *Crackers* was the term for people that performed attacks on computer systems

• Nowadays, “hacker” has a negative connotation

• Note, though, the distinction between *white hat* and *black hat* hackers

• These days, purely criminal organizations are more and more active

• Don’t try this at home
Dangers in change, and in abstraction

• Change is a big problem

• Even if (you think) you understand the implications of a change, it is easy to get it wrong

• Abstraction is very useful to understand complex systems

• However, security implications are often so detail-dependent that hiding the details can be a big problem

• Sometimes, the abstraction does not correspond to the actual implementation

• It is very important to be clear about the threats; these often drop from view in abstract models
Abstraction threats: Characters

• You want to give access only to /A/B/C, and your application appends the input into /A/B/C/input

• Attacker enters ../../../etc/passwd

• Input validation should be used

• UTF-8 specifies %c0%af=’/’

• An old Microsoft IIS accepted this, so that [IP]/scripts/../../../winnt/system32/ decoded to C:\winnt\system32
Abstraction threats: Characters

• UTF-8 specifies `%c0%af=`’/

• An old Microsoft IIS accepted this, so that
  [IP]/scripts/..%c0%af../winnt/system32/ decoded to
  C:\winnt\system32

• There is a further twist, since the decoding is to binary and then a
  further string decoding

• The string [IP]/scripts/..%25%32%66../winnt/system32/
  decodes to [IP]/scripts/..%2f../winnt/system32/ which
  decodes to [IP]/scripts/../../winnt/system32/

• Decoding UTF-8 is translation between levels of abstraction
Abstraction threats: Integers

- 8-bit integers: $255 + 1 = 0$
- 8-bit signed integers: $127 + 1 = -128$, $-128 / -1 = -128$
- Type confusion: $255$ (unsigned) $= -1$ (signed)
- The comparison
  
  ```
  if (size < sizeof(buf))
  ```
  might be true: if size is signed and you assign a large value, it might turn negative

- UNIX once contained programs that first checked that the UID is not zero (\(=\)root), and then truncated the UID to an unsigned short
Abstraction threats: Integers

- Basic problem: computer integers are not mathematical integers
- \( b \geq 0 \not\Rightarrow a + b \geq a \)
- Use unsigned integers, watch out for integer overflow
- Turn on compiler warnings for signed-unsigned comparison
Canonicalization

- Filenames have several different but equivalent representations
- Dotless IP have 32 bits: \( a.b.c.d = 2^{24}a + 2^{16}b + 2^8c + d \)
- Symbolic (soft) links give more equivalent representations
- Some systems have case-insensitive filenames (for example, consider old Apache on HFS+)
- Perform access-control decisions in one unique canonical representation
Memory management

- Buffer overruns
  - Stack overruns
  - Heap overruns
- Double-free vulnerabilities
Stack overruns

```c
void myfunction(int a, int b)
{
    char x[20];
    ...
}
```

![Diagram showing stack and heap with labels: input to b, input to a, return address, saved fp, buffer for x, stack, heap, library]
void myfunction(int a, int b)
{
    char x[20];
    ...
}

Stack overruns

void myfunction(int a, int b)
{
    char x[20];
    ...
}
Stack overruns

```c
void myfunction(int a, int b)
{
    char x[20];
    ...
}
```

![Diagram showing stack and heap structures with labels: input to b, input to a, bad ret addr, saved fp, buffer for x, stack, heap, library.]
The 1980s — the era of personal computers

- The “Morris worm” of 1988 infected 5-10% of all machines connected to the internet
- Used a buffer overrun in the *fingerd* daemon of VAXes running BSD Unix
- Perpetrator sentenced to $10000 fine and 400 hours community service

```
push1 $68732f push ’/sh, <NUL>’
push1 $6e69622f push ’/bin’
mov1 sp, r10 save stackp in r10 (string beginning)
push1 $0 push 0 (arg 3 to execve)
push1 $0 push 0 (arg 2 to execve)
push1 r10 push string beginning (arg 1 to execve)
push1 $3 push argc
mov1 sp, ap set argv to stackp
chmk $3b perform ’execve’ kernel call
```
Buffer overruns, and “smashing the stack”

- Basic weakness: programmers are often careless about checking the size of arguments
- An attacker passes a long argument can find that some of it is treated as executable code rather than data

• About half of the CERT bulletins are (were 2008) of this kind, this is slowly decreasing
Buffer overruns, and “smashing the stack”

• Basic weakness: programmers are often careless about checking the size of arguments

• An attacker passes a long argument can find that some of it is treated as executable code rather than data

About half of the CERT bulletins are (were 2008) of this kind, this is slowly decreasing
Heap overruns

- These are harder to perform since it is more difficult to predict where the buffer is in relation to the target
- Target is usually pointers
  - Pointers to open files
  - Pointers to functions (requires executable heap)
- Effect can be crash rather than break, but with enough attempts (large vulnerable user base), the attack will eventually result in a break
Double-free vulnerabilities

- In a double-free vulnerability, the OS itself is lured into writing into the target memory location.
- If memory is free’d but the pointer not zeroed, it could be free’d again.
- The function `malloc` allocates a chunk of memory.
- The function `free` gives it back to the system.
- Free memory is kept in a double-linked list.
- There is a mechanism to join chunks back together, and this is the weakness.
Double-free vulnerabilities

• In a double-free vulnerability, the OS itself is lured into writing into the target memory location

• If memory is free’d but the pointer not zeroed, it could be free’d again

• The function malloc allocates a chunk of memory

• The function free gives it back to the system

• Free memory is kept in a double-linked list

• There is a mechanism to join chunks back together, and this is the weakness
Double-free vulnerabilities

- In a double-free vulnerability, the OS itself is lured into writing into the target memory location.

- If memory is free’d but the pointer not zeroed, it could be free’d again.

- The function `malloc` allocates a chunk of memory.

- The function `free` gives it back to the system.

- Free memory is kept in a double-linked list.

- There is a mechanism to join chunks back together, and this is the weakness.
Double-free vulnerabilities

- In a double-free vulnerability, the OS itself is lured into writing into the target memory location.
- If memory is free’d but the pointer not zeroed, it could be free’d again.
- The function `malloc` allocates a chunk of memory.
- The function `free` gives it back to the system.
- Free memory is kept in a double-linked list.
- There is a mechanism to join chunks back together, and this is the weakness.
Double-free vulnerabilities

• In a double-free vulnerability, the OS itself is lured into writing into the target memory location

• If memory is free’d but the pointer not zeroed, it could be free’d again

• The function `malloc` allocates a chunk of memory

• The function `free` gives it back to the system

• Free memory is kept in a double-linked list

• There is a mechanism to join chunks back together, and this is the weakness
Distinguishing data and code: scripting

- The importance of sanitizing input cannot be underlined too much

    ```bash
    #! /bin/bash
cat $1 | mail $2
    ```

- Call this with the following

    ```bash
    foo thefile 'nobody@home | rm -rf /
    ```

- Or rather, don’t!
SQL injection

Hi, this is your son's school. We're having some computer trouble.

Oh, dear - did he break something?

In a way -

Did you really name your son Robert'); DROP TABLE Students;-- ?

Oh, yes. Little Bobby Tables, we call him.

Well, we've lost this year's student records. I hope you're happy.

And I hope you've learned to sanitize your database inputs.
Race conditions

- Strange things can happen when multiple processes or threads access the same data

- In CTSS (a time-sharing OS from the 60s), a user found that the “message of the day” contained the password file
  - CTSS was designed for low memory, and had a tempfile for the editor named “SCRATCH” in the home directory
  - This was no problem since the home directory was writable only by the owner
  - Later, the system user was allowed to be used by several people
  - One edits “message of the day”, another edits the password file...

- Another example of TOCTTOU (if in different clothes)
- Can be prevented by using file locking
- You can still find this in modern systems as unsafe tempfile handling
Prevention: Hardware

- An example is Intel’s Itanium that has a separate register for the return address.
- A more extreme solution is to put the return address in a separate Secure Return Address Stack.
- This kind of hardware protection does not need recompilation.
- But more extensive changes (to processor instructions, for example) may need changes in multi-threaded programs.
Prevention: Modus operandi

- A non-executable stack stops certain attacks
- Software that requires executable stack will stop working
- It is to the attacker’s advantage if memory usage is predictable
- Address space layout randomization can prevent this
- BSD has used this to prevent argv[] attacks
- Windows uses this in system libraries as a defence against return-to-libc attacks
Prevention: Safer functions

- C is infamous for its string handling functions, say strcpy, sprintf, or getc
- For strcpy, the result is undefined if strings are not null-terminated
- There is no check if the destination buffer is long enough
- The function strncpy is better, since there is a count argument for the longest string length
- But watch out! this does not put a null at the end of the string
- Also watch out for integer overflows
- Perhaps use bstrlib?
Prevention: Filtering

• Whitelisting is the safer option
• Blacklisting is more difficult to get right
  • You must know about all dangerous inputs
  • ... in all encodings (UTF-7 has been used in XSS attacks)
  • “Helpful” system components may trip you, say converting a foreign character into < or ’
• Filtering is difficult and complex
Prevention: Type safety

- There are programming environments (and compilers) that check unsafe usage of types
  - *Dynamic* type checking checks at runtime, and slows the program down
  - *Static* type checking does the checking in advance, at compile time; this requires more complicated checking, but does not decrease performance at runtime

- What is often ensured is memory integrity

- We would want to get execution integrity; may be difficult to specify exactly what this means
Detection

- *Canaries* are memory elements used to detect unwanted changes to memory at runtime
- *Code inspection* by hand is slow and error prone, but does help somewhat
- *Automated code inspection* uses an expert system with known weaknesses
- Security *testing* does not need the source code, but may use the specification of allowed inputs and expected outputs
  - Random inputs are not so useful, since the attacker should choose the distribution
  - Common attacks needs to be tested; many examples are listed in the book
Mitigation: Least privilege

- Be sparing with requiring privileges to run the code
- Do not give users more rights than needed
- Drop rights immediately when possible
- Do not activate options you do not need
Reaction: keeping up to date

\[ t_d \ldots \text{time of disclosure} \]
\[ t_p \ldots \text{time patch released} \]
\[ t_a \ldots \text{time attack script released} \]