Software Exploitation

Shuai Wang

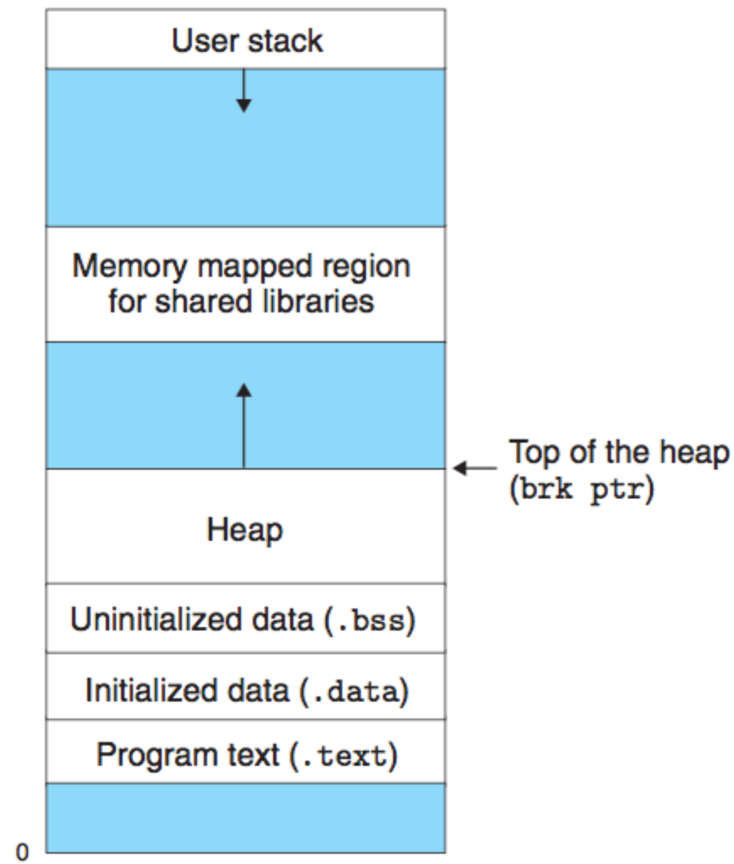
香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Software Memory Vulnerabilities

- A memory error allows a program statement to access memory beyond allocated
  - Buffer overflow ← talked before
  - Use-after-free ← heap
  - Double-free ← heap
  - Type confusion ← heap
  - Format string ← stack
- SQL Injection ← database input validation
  - Mostly due to input validation check failure

Heap vs. Stack

- Linux standard memory layout but Mac/Windows are very similar.

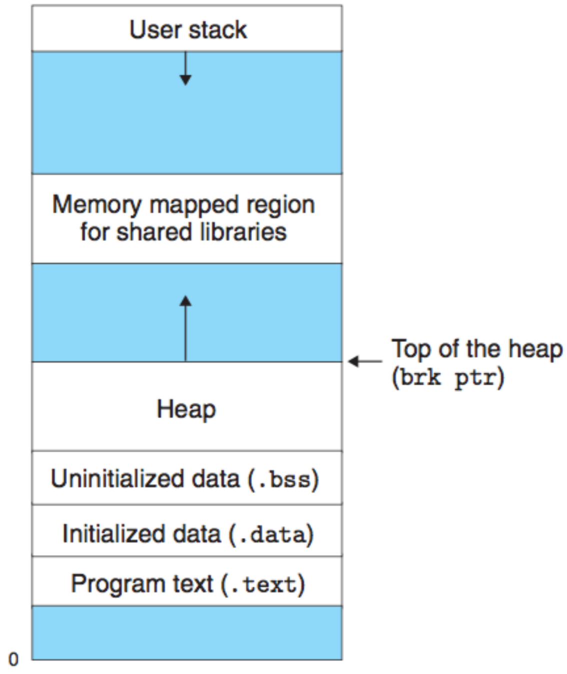


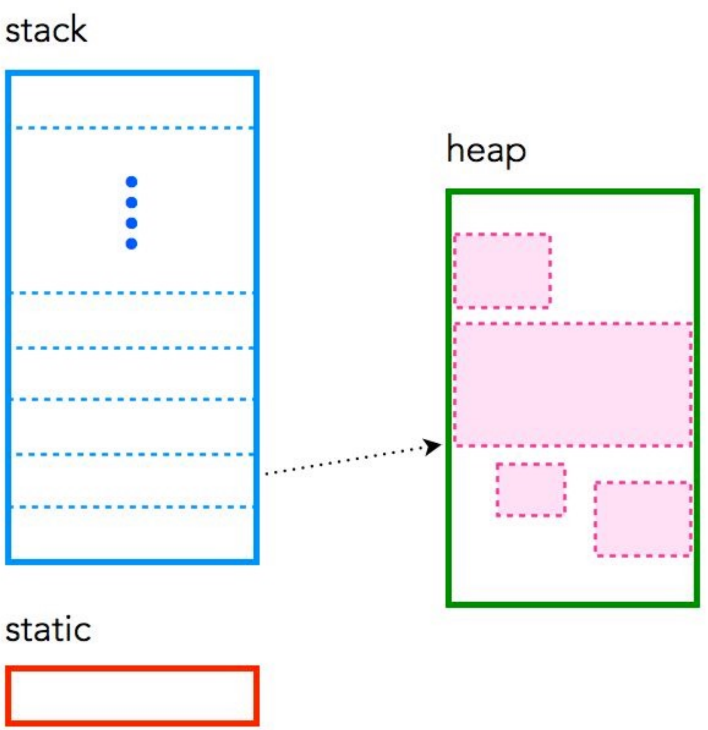
Heap Memory

- **heap** is the portion of memory where *dynamically allocated* memory resides
  - C language: `malloc`
  - C++ language: `new`
- Memory allocated from the heap will remain allocated until one of the following occurs:
  - Program terminates
  - Software calls `free` (for C) or `delete` (for C++)

Heap Memory

- Linux standard memory layout but Mac/Windows are very similar.

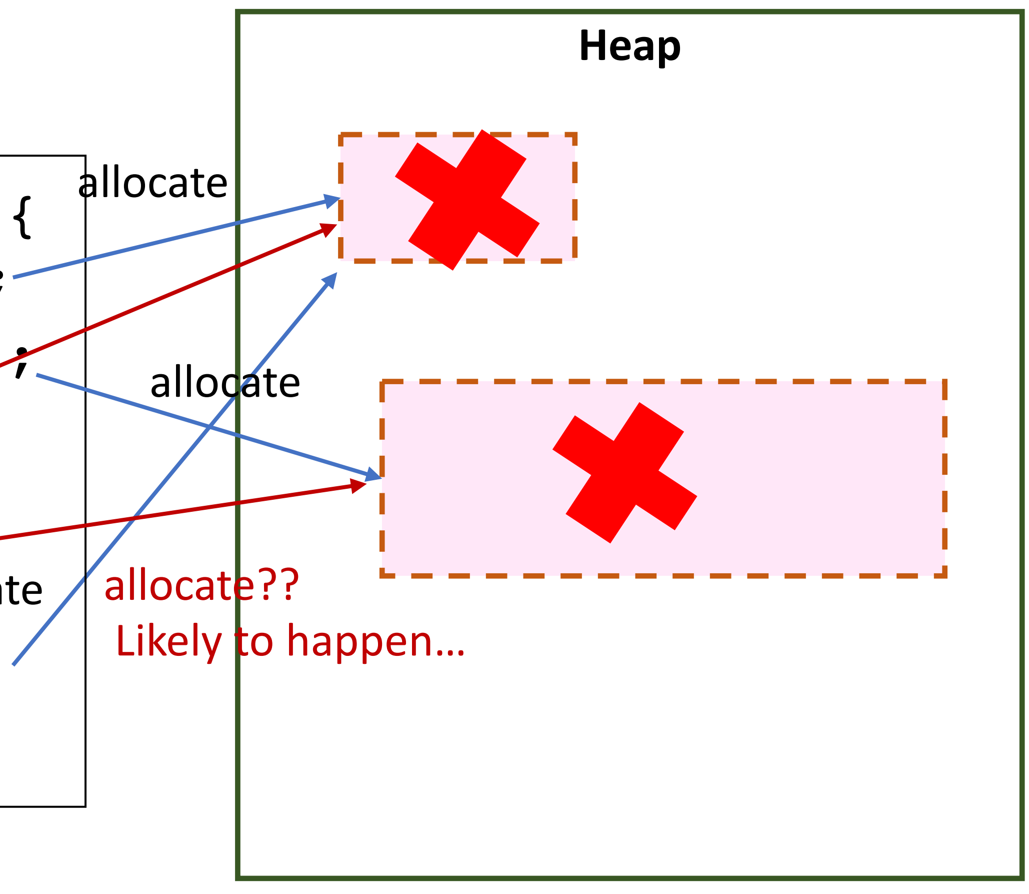




Heap Memory Allocation/Deallocation

```
void func(int a, int b){
   int *p = new int[10];
   int *q = new char[40];

   delete p;
   delete q;
   int *x = new int[10];
}
```

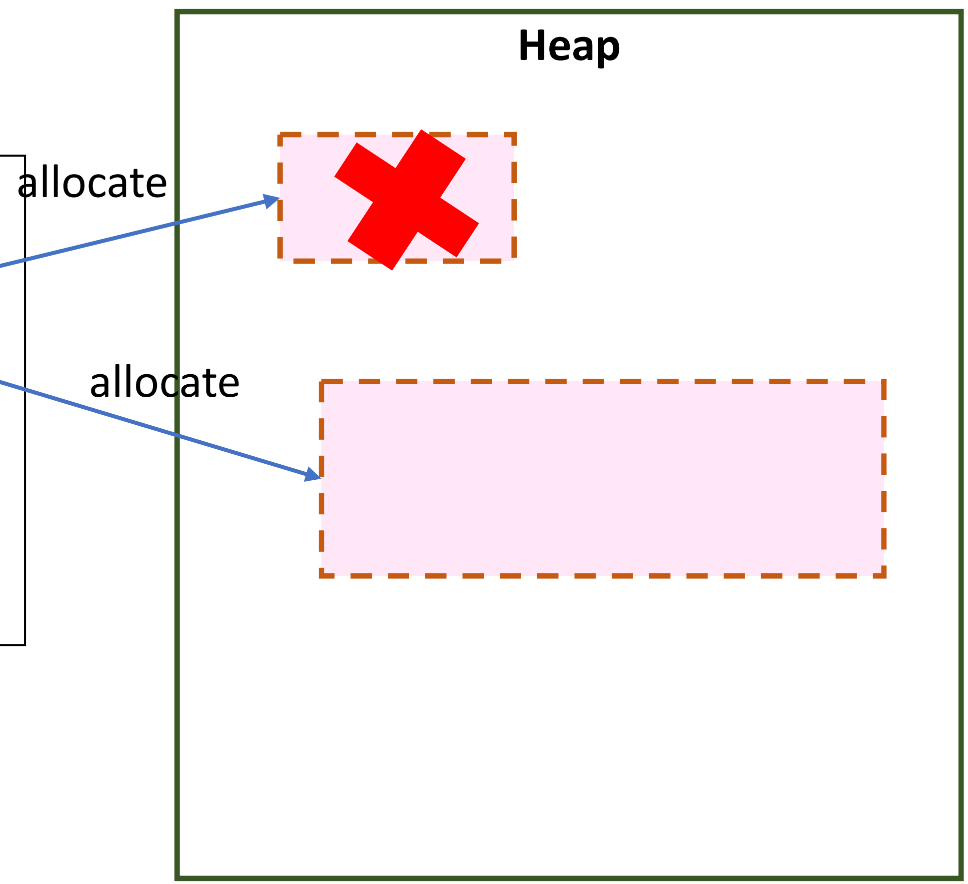


Programmers need to take care of heap memory

Heap Memory Leak

```
void func(int a, int b){
   int *p = new int[10];
   int *q = new char[40];

   delete p;
}
```

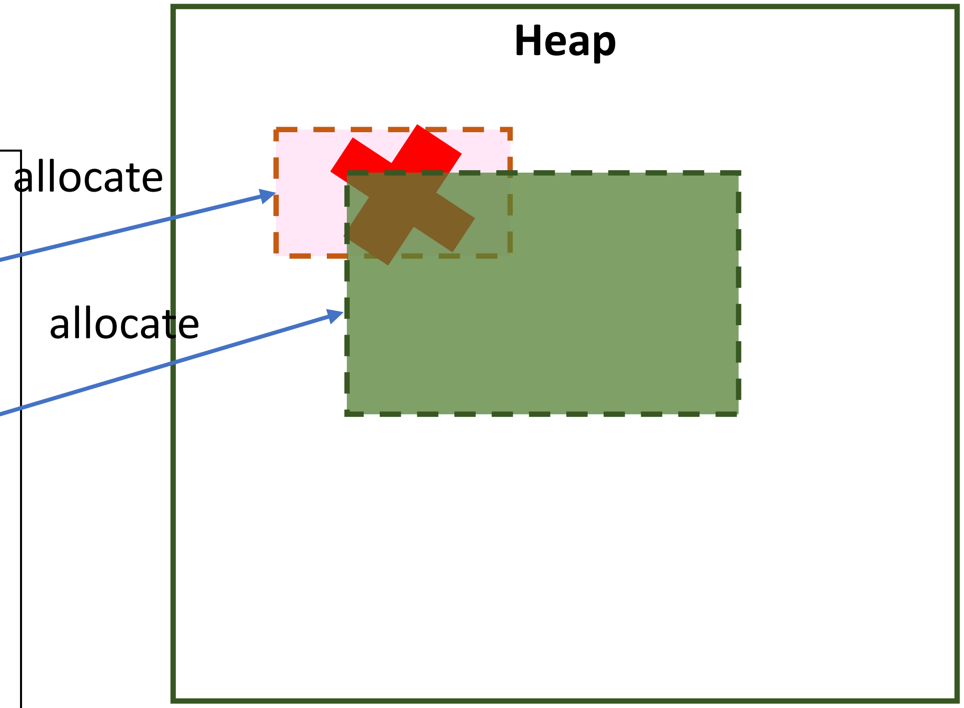


Resource abuse bug can lead to crash/lost response → availability

Use After Free

- Flaw: Program frees data on the heap, but then references that memory as if it were still valid

```
void func(int a, int b){
   int *p = new int[10];
   delete p;
   int *q = new char[90];
   *p = 50;  <--- ?
}
```



simplified!

Use After Free

```
struct A {
    void (*fnptr)(char *arg);
    char buffer[40];
};

struct B {
    int B1;
    int B2;
    char info[32];
};
```

```
x = (struct A *)malloc(sizeof(struct A));
free(x);
y = (struct B *)malloc(sizeof(struct B));

y->B1 = 0xDEADBEEF;
x->fnptr(buf);
```

possibility of executing arbitrary code

```c
struct auth {
  char name[32];
  int auth;
};

struct auth *auth;
char *service;

int main(int argc, char **argv)
{
  char line[128];

  while(1) {
    printf("[ auth = %p, service = %p ]\n", auth, service);

    if(fgets(line, sizeof(line), stdin) == NULL) break;

    if(strncmp(line, "auth ", 5) == 0) {
      auth = malloc(sizeof(auth));
      memset(auth, 0, sizeof(auth));
      if(strlen(line + 5) < 31) {
        strcpy(auth->name, line + 5);
      }
    }
    if(strncmp(line, "reset", 5) == 0) {
      free(auth);
    }
    if(strncmp(line, "service", 6) == 0) {
      service = strdup(line + 7);
    }
    if(strncmp(line, "login", 5) == 0) {
      if(auth->auth) {
        printf("you have logged in already!\n");
      } else {
        printf("please enter your password\n");
      }
    }
  }
}
```

A server can never be logged in?

You don't know the pwd but can you somehow execute "you have logged in already!" ??

1. create a "user"

4. cleanup and remove "user"

3. specify a "service" for use

2. "login"

strdup: allocate a chunk of mem on heap

https://exploit.education/protostar/heap-two/; only a proof-of-concept (PoC) so don't take it too seriously.

```c
struct auth {
  char name[32];
  int auth;
};

struct auth *auth;
char *service;

int main(int argc, char **argv)
{
  char line[128];

  while(1) {
    printf("[ auth = %p, service = %p ]\n", auth, service);

    if(fgets(line, sizeof(line), stdin) == NULL) break;

    if(strncmp(line, "auth ", 5) == 0) {
      auth = malloc(sizeof(auth));
      memset(auth, 0, sizeof(auth));
      if(strlen(line + 5) < 31) {
        strcpy(auth->name, line + 5);
      }
    }
    if(strncmp(line, "reset", 5) == 0) {
      free(auth);
    }
    if(strncmp(line, "service", 6) == 0) {
      service = strdup(line + 7);
    }
    if(strncmp(line, "login", 5) == 0) {
      if(auth->auth) {
        printf("you have logged in already!\n");
      } else {
        printf("please enter your password\n");
      }
    }
  }
}
```

A server can ~~never~~ be logged in with use-after-free bug!

1. create a “user”

2. cleanup and remove “user”

3. specify a “service” for use

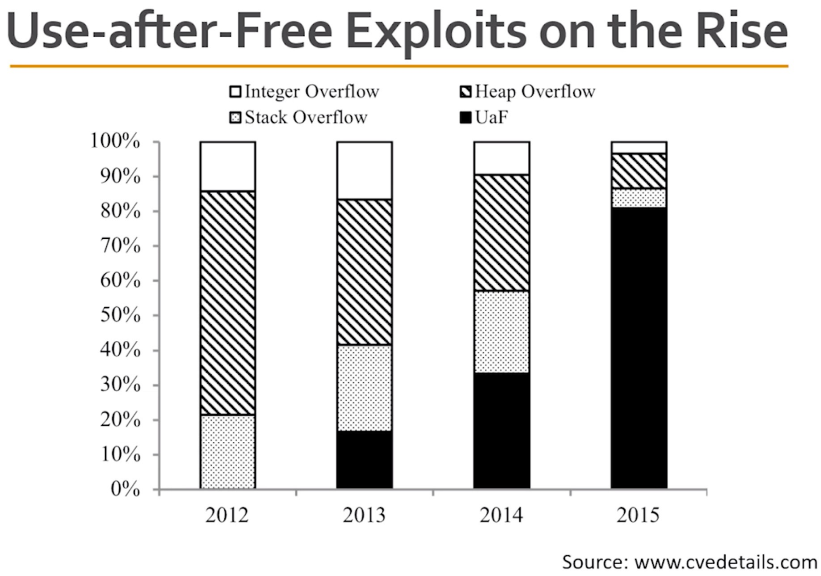
4. “login”

strdup: allocate a chunk of mem on heap

https://exploit.education/protostar/heap-two/; only a proof-of-concept (PoC) so don’t take it too seriously.

Use After Free

- Flaw: program frees data on the heap, but then references that memory as if it were still valid
- Accessible: Adversary can control data written using the freed pointer
- Become a popular vulnerability to exploit



Why detect use-after-free (with fuzz testing) is difficult? complexity + saliency
Your reading materials today…

Prevent Use After Free

- What can you do (not too complex)?
  - You can set all freed pointers to NULL
  - Then, no one can use them after they are freed
  - Use smart pointers or other strategies
    - optional reading materials today

Related Problem: Double Free

```
main(int argc, char **argv)
{
        …
        buf1R1 = (char *) malloc(BUFSIZE2);
        buf2R1 = (char *) malloc(BUFSIZE2);

        free(buf1R1);
        free(buf2R1);

        buf1R2 = (char *) malloc(BUFSIZE1);
        strncpy(buf1R2, argv[1], BUFSIZE1-1);

        free(buf2R1);
        free(buf1R2);
}
```

Free the R1 buffers

Allocate a new buffer R2 and supply data

Free the R1 again, which uses R2 data as metadata

Then, free R2 which uses really messed up metadata

Double Free

- In general, double free messes up the heap memory region
  - Can be exploited by deliberately constructing the memory allocation/deallocation chain
- Prevent Double Free
  - Save yourself some headache by setting freed pointers to NULL

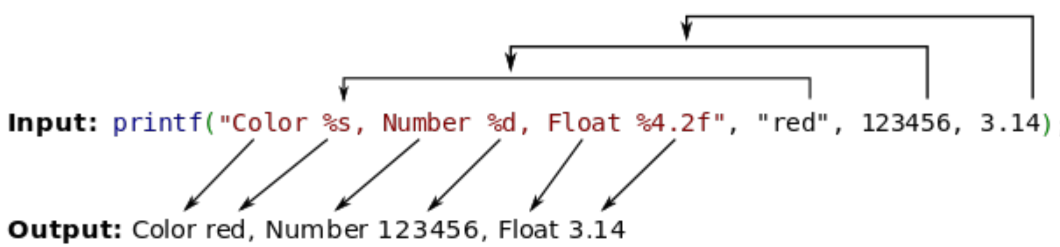
Format String Vulnerability

- Who ever uses `printf` in their programs?

```
printf ("This class is %s\n", string);
```

  - Or C++ `Boost::format` or C++ `vsnprintf` or C++ 2020 specs…

In some cases, printf or any functions use format string as inputs can be exploited



Format String Vulnerability

- `printf` takes a format string and an arbitrary number of subsequent arguments
  - Format string determines what to print
    - Including a set of format parameters
  - Arguments supply input for format parameters
    - Which may be values (e.g., %d) or references (e.g., %s)
  - An argument for each format parameter

```
printf ("This class is %s\n", string);
```

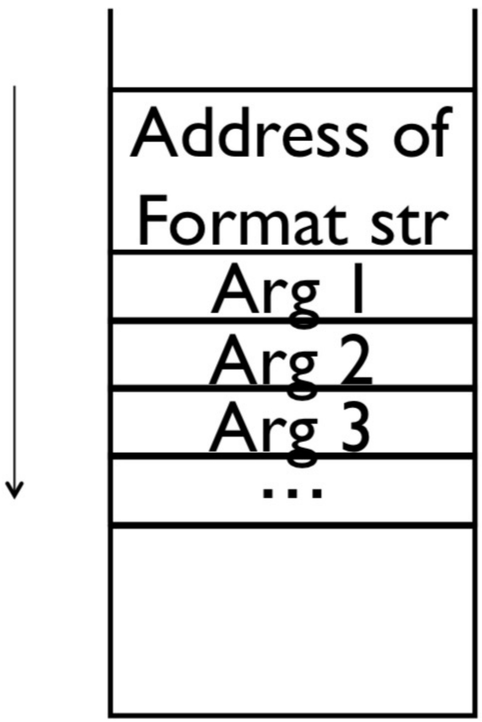
Format String Vulnerability

- Who uses `printf` in their programs?
  - In some cases, printf can be exploited
- As usual, arguments are retrieved from the stack
  - What happens when the following is done?

```
printf("%s%s%s%s");
```

- Traditionally, compilers do not check for a match between arguments and format string
  - So, `printf` would print "strings" using next four values on stack as string addresses – whatever they are

Printf and Stack



- Remember these are parameters to a function call
- So, the function expects them on the stack
- Printf will just start reading whatever is above the format string address

Format String Vulnerability

- printf can take a variable as an argument – treated as a format string
  - If an adversary can control this argument, they can direct printf to access that memory – “%s%s%s…”

You might be surprised but I see this on Twitter last summer….

Format String Vulnerability

- An interesting format parameter type – %n
  - "%n" in a format string tells the printf to write the number of bytes written via the format string processing up to that point to an address specified by the argument

```c
#include <stdio.h>

int main()
{
  int val;

  printf("blah %n blah\n", &val);

  printf("val = %d\n", val);

  return 0;

}
```

```
blah  blah
val = 5
```

output

Enable arbitrary memory write by controlling the format string and the second argument.

Prevent Format String Vulnerability

- Preventing format string vulnerabilities means limiting the ability of adversaries to control the format string, how?
  - Hard-coded strings w/ no arguments – when you can
  - Hard-coded format strings at least – no `printf(arg)`
  - Do not use %n

SQL Injection

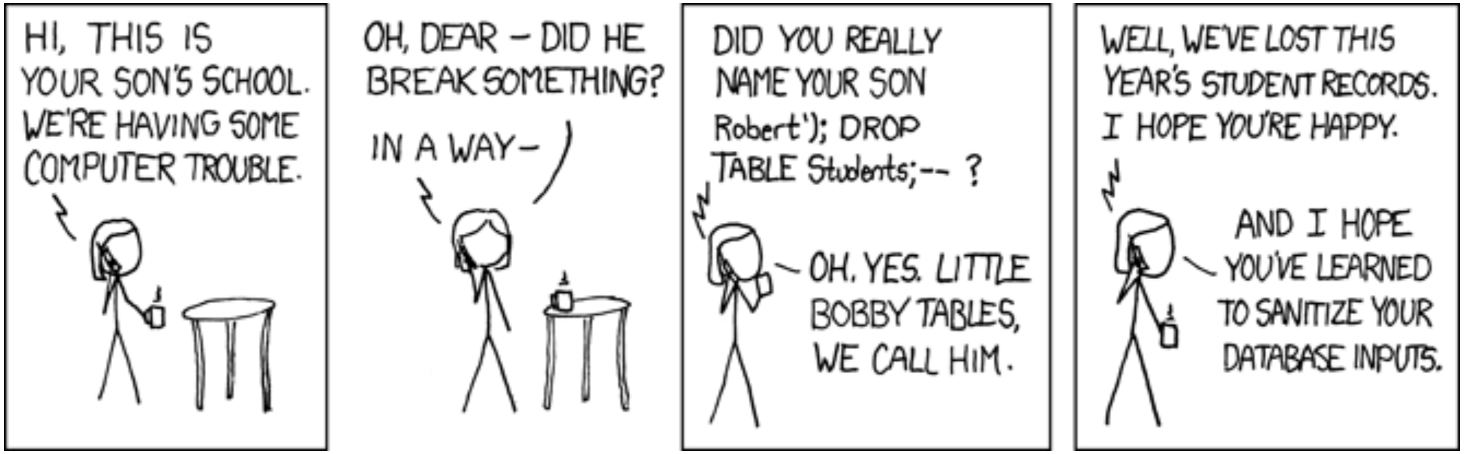
- **SQL injection**:
  Causing undesired SQL queries to be run on your database.
  - Often caused when untrusted input is pasted into a SQL query

    `PHP: "SELECT * FROM Users WHERE name='$name';";`
  - specify a user name of: **`x' OR 'a'='a`**

    `SELECT * FROM Users WHERE name='x' OR 'a'='a';`

SQL Injection

- **SQL injection**:
  Causing undesired SQL queries to be run on your database.
  - Often caused when untrusted input is pasted into a SQL query




```
SELECT * FROM Users WHERE name=('Robert'); DROP TABLE Students;
```

http://www.circleid.com/posts/20130325_sql_injection_in_the_wild/

Best defence

- If possible, use bound variables with prepared statements
  - Many libraries allow you to bind inputs to variables inside a SQL statement
  - PERL example (from http://www.unixwiz.net/techtips/sql-injection.html)

```
$sth = $dbh->prepare("SELECT email, userid FROM members WHERE
    email = ?;");
$sth->execute($email);
```

How does this prevent an attack?

- The SQL statement you pass to prepare is parsed and compiled by the database server.
- By specifying parameters (either a ? or a named parameter like :name) you tell the database engine what to filter on.
- Then when you call execute the prepared statement is combined with the parameter values you specify.
- It works because the parameter values are combined with the compiled statement, not a SQL string.
  - SQL injection works by tricking the script into including malicious strings when it creates SQL to send to the database.

More Defenses

- Check syntax of input for validity
  - Many classes of input have fixed languages
    - Email addresses, dates, part numbers, etc.
    - Verify that the input is a valid string in the language
    - Some languages allow problematic characters (e.g., ‘*’ in email); may decide to not allow these
- Have length limits on input
  - Many SQL injection attacks depend on entering long strings

Even More Defenses

- Scan query string for undesirable word combinations that indicate SQL statements
  - INSERT, DROP, etc.
  - If you see these, can check against SQL syntax to see if they represent a statement or valid user input
- Limit database permissions
  - If a user only reads the database, grant only read permissions

Saltzer and Schroeder's Principles of Secure Design

- 3) Least Privilege

A subject should only be given the minimum necessary privileges for completing its task.