CISC 327: Software Quality Assurance

Lecture 19–1 (28 in 2017)
Introduction to Insecurity
“19–1” say what

• My next surgery is scheduled for Nov. 19th
• No one else has taught from my security slides
• I really like my security material
• Scrambling the schedule to cover security earlier
• inspection; metrics; security
  ⇒ (code smells); security; inspection; metrics

• **Mini-Exam #3:**
  will not cover inspection, **will** cover security
  (and mutation/continuous/regression testing)

• **Mini-Exam #4:**
  will not cover security, **will** cover inspection
  (and metrics)
But first...

- E2 marked
- Average slightly lower than E1
- A “feature” of non-memorization-based testing is that you can have the right idea but flub the execution
THE MANAGEMENT RESPECTFULLY WISHES TO CLARIFY THE FOLLOWING POINTS

• **Completeness criterion** for systematic testing: knowing when you have enough test cases, **not** when you have run the tests

• Even though my intended answers for parts (ii) and (iii) were nearly identical, you could get full marks by saying all the right things somewhere

• **Decision coverage**: not one test per decision

• The white box/black box checkboxes were worth only 1/3 mark each
THE MANAGEMENT RESPECTFULLY WISHES TO CLARIFY THE FOLLOWING POINTS

• On Q2, a lot of redundant tests:
  – "(TMC)" on the front: "too many cases"
THE MANAGEMENT RESPECTFULLY WISHES TO CLARIFY THE FOLLOWING POINTS

• On Q2(c), exhaustive output coverage, many people either missed the word “output” or didn’t make the connection from input to output (you can derive the number of possible outputs by reasoning about the number of inputs, but in general, “infinite” inputs alone does NOT imply “infinite” outputs: String.isEmpty)

• Many people claimed that the set of strings is infinite, which is not quite true
Uphill in the snow, both ways, finite

140K on each side of a floppy

128K RAM = about $2^{(1 \text{ million})}$ possible RAM states

(All About Apple Museum, CC BY-SA 3.0)
Uphill in the snow, both ways, finite

128 \textit{bytes} \text{ RAM} = \text{about } 2^{1024} \text{ possible RAM states}

Still a lot of strings to test...but not infinite.
THE MANAGEMENT RESPECTFULLY WISHES
TO CLARIFY THE FOLLOWING POINTS

• On Q3(c):
  – Many answers trying to litigate the question of whether black-box unit testing is actually black-box testing
  – It's sometimes called gray-box testing
  – Some answers had the form "NO you can't do it! but you can do white-box testing by using a test harness and writing a stub!"
  – Apart from the gray-box issue, some confusion about when you can do black-box testing
    • "black-box testing does not require code"
      – it doesn’t require source code to design the tests, but you need executable code to run the tests
    • "black-box testing only works if you don't have the code, if you have the code you're doing white-box testing"
      – black-box test cases can be designed without having any code, and black-box tests can be run without access to the source code (but you need an executable), but black-box testing is still possible if you have source code
THE MANAGEMENT RESPECTFULLY WISHES TO CLARIFY THE FOLLOWING POINTS

• On Q4, a lot of redundant tests:
  – I decided it was equally bad to write one extra case, "(TM)", as several, "TM": if you're going to ignore the instruction to write only as many cases as needed, e.g. by doing input partitioning in addition to output partitioning, you should actually do input partitioning properly rather than throwing in one extra case
THE MANAGEMENT RESPECTFULLY WISHES TO CLARIFY THE FOLLOWING POINTS

• On Q4, a lot of redundant tests
  – "TM" or "(TM)" on the front: "too many"
  – The entire 0 = wildcard business was irrelevant to the answer, because it doesn't affect the output partitions!
  – The question wants output partitions, not a thorough set of black box test cases
  – Each testing method has different strengths and weaknesses: know what each one does, and what it doesn’t do
  – Thorough testing often involves applying multiple testing methods
THE MANAGEMENT RESPECTFULLY WISHES TO CLARIFY THE FOLLOWING POINTS

• Q5(b)
  – Generally good but suffered from a lack of specificity (writing “x < 0” instead of –1, etc.)
  – I respect the desire to think abstractly about the equivalence class of input, but that is not an actual test case that you could run (unless you’re using abstract interpretation technology!)
Not someone else’s slides

• I prefer to treat a few topics in detail, rather than give a wide-ranging overview
• Not a perfect solution but there is no perfect solution—as with every other section of 327, security really deserves an entire course
Security

• What is (computer) security?
  – Preventing “theft and damage to...hardware, software or information, as well as...disruption or misdirection of the services they provide” (Wikipedia)
EVIL TIME

ALL THE TIME
Outline

• Some definitions
• Technical security and user security
• Light overview of security
• Exploit: Buffer overruns
  – The Morris worm
• Exploit: Walking through the open front door
  – The Mac OS resource fork
Unauthorized Access

• A user causing a program to carry out an action that the user is not allowed to do
  – Showing the user someone else’s private information
  – Crashing a computer system
    • Borderline case: Radio Shack in the late 1980s
    • Random customers could access the systems but were probably not supposed to use them for certain purposes
  – A bank payment not authorized by the bank account’s owner
  – Using an unsecured WiFi router for internet access
  – Turning off loud TVs in airports
“Social Engineering”

• As with “quality parameters” (user vs. technical), much of computer security is about humans, not machines
  – having no key or ID card to access a building, but appealing to a stranger’s courtesy by tailing them and “allowing” them to hold the door for you
  – persuading customer support to reset a password by giving them publicly available “private” information
  – pretending to be your ISP, employer, or the FEDERAL BUREAU OF INVESTIGATION in an email
    • “Nigerian scammers”
  – relying on user unwillingness to use different passwords to “escalate” a theft of one password to access all the user’s accounts
“Social Engineering”

• As with the User Quality Parameters at the start of 327, our coverage of usable security will only acknowledge that the human side exists, and matters!
  – Therac-25 failed on the technical side (buggy, hard to test) and the user side (terrible error messages)

• We will focus on the technical side of security
Why security is hard

• The damage caused by software faults is not always cumulative

• The deaths caused by Therac-25 could probably have been avoided if either of the following had been done
  – software design that allowed thorough testing
  – better error messages that advised the operator that the machine settings made no sense, and so the dosage level could not be guaranteed
Why security is hard

• Usually, the damage caused by a fault, or even a series of faults, is not necessarily cumulative

• The *Mars Climate Orbiter* crashed (–US$327M) due to a succession of failures:
  – the software mixed up SI units (newtons) and Imperial units (pound-forces), and fired the thrusters incorrectly
  – resulting in the predicted position differing from the actual position
  – the software could conceivably have noticed this
  – two operators noticed this, but their “concerns were dismissed”
Security is really hard

• But insecurity is often cumulative
• A single system vulnerability is enough
• If there are 100 ways to steal a password, and you prevented 99 of them, you failed
• Security is an exam where you have to get every question right to pass
Any kind of defect could be a security hole

```java
itemsEqual = true; // oops, should be ‘false’
...
if (itemsEqual) {
    ...
}
```

- **Case #1:** The items being compared are dates (to display an “easter egg” on the programmer’s birthday), and consequently, the easter egg is always displayed


Any kind of defect could be a security hole

itemsEqual = true;  // oops, should be ‘false’
...
if (itemsEqual) {
   ...
}

• **Case #2:** The items being compared are password strings, and consequently, any password works
How you could steal a password...

• User side
  – Watch the user’s fingers
  – I tried your cat’s name
  – I tried your favorite bands
  – I have the password to your Shell account
  – Barcelona,
    “I Have The Password to Your Shell Account” (2000)
  – “Rubber-hose decryption”
How you could steal a password...

• Technical side (hardware/firmware)
  – See the DEFCON talk linked in Lecture 24
  – “Backdoor keylogger” in the keyboard circuitry
  – ...or in the CPU
  – ...or in the BIOS
How you could steal a password...

• Technical (software) side
  – Read the password file
    • the password file is itself encrypted, so...
  – Steal the password file
    • if you can copy the password file to your own computer, you can try to crack the encryption at your leisure
  – Install a keylogger that records all keystrokes
    • installing the keylogger is usually its own “subproblem”, though not on the early Mac OS...
How you could steal a password...

• Technical (software) side
  – Before ‘ssh’, we used ‘rsh’ and ‘telnet’, which didn’t encrypt passwords
  – So if the password was being sent over the network, you could read it if...
    • you worked for the user’s ISP
    • you worked for any networking company (“the backbone”) on any intermediate server
    • you had already stolen the password of someone who worked for either of those
  – Since around 2000,
    we use ‘ssh’, which encrypts the connection
How you could steal a password...

• Technical (software) side
  – Much of this discussion goes double for passwords or private information entered on websites
  – Before secure HTTP, data entered into web forms was sent in cleartext
  – The OpenSSL library implements encryption for web traffic
    • “Heartbleed” (marketing is 50% of security?) was a major security flaw, sort of a buffer overrun exploit but not in the usual way
How you could steal a password...

• Technical (software) side
  – Get a job at the OS vendor, and put a backdoor in the operating system
  – “Wait,” you say, “wouldn’t someone notice?”
  – What if you put a backdoor in the compiler?
    • what if you put a backdoor in the programs you use to inspect the object code of the compiler?
      – .
      » ...
  • Ken Thompson, Reflections on Trusting Trust (1984)
How you could steal a password...

• Technical (software) side
  – **Timing attacks:** Measure the time used by the “try password” system call, which could be longer for long passwords (gives an idea of password length) and might also depend on the contents
    • can try to compensate by doing a pseudo-random quantity of busywork to the code for that system call
      – what if the attacker has compromised the pseudo-random number generator?
Buffer overrun exploits
Buffer overrun exploits

• If you took CISC 221, some of this may look familiar
• I can’t fit 221 into this lecture, and I never learned x86 assembly, so we’ll have to skip a lot of details
• This discussion will focus on compiled programs in C, but goes for most other low-level languages too
Buffer overrun exploits

• When you call a function in compiled C, a bunch of stuff happens to the stack:
  • the caller pushes arguments
  • the caller pushes its return address, so the called function (“callee”) knows where to jump when it’s finished
  • the callee grows the stack to store its local variables
  • (traditionally, a CPU stack grows towards smaller addresses: the “bottom” of the stack might be 0x00980000, a larger address than 0x0097EE1C)
Buffer overrun exploits

• This partitions the stack into sections, each containing a function’s arguments, return address, and local variables

• Each section is a stack frame

• Just before main() is called in bufcopter.c, there will already be some stack frames for the C runtime library and maybe the operating system
Buffer overrun exploits

• Then, main() calls sum_integers()
• Inside sum_integers(), the stack will be as shown in the beautiful ASCII art

```
STACK
POINTER
  `----> _______________
pushed> | i      | << stack frame for sum_integers()
by  > | sum    |
sum_int  buf[0]  | local buffer, contains 0
egers    buf[1]  |     1
> | .    |
> | .    |
> | .    |
> | buf[127] | 127
|.............|
++-> | return address | sum_integers's return address (points inside main())
|    | m       | sum_integers's 1st (only) argument
|    |         |                      
pushed> | z3   | << stack frame for main()
by  > | z2    | \______ == buf[134] (buf has 64-bit longs, z2 and z3 are 32-bit ints)
main > | z1    |   /   (on my laptop; your results may vary!)
|.............|
pushed> | return address | main's return address (points inside C runtime library)
by  > | argv  | main's 2nd argument
caller| argc  | main's 1st argument
|        |        |         
```
Buffer overrun exploits

- Then, main() calls sum_integers()
- Inside sum_integers(), the stack will be as shown in the beautiful ASCII art
Buffer overrun exploits

- If `sum_integers()`’s return address gets overwritten somehow, `sum_integers()` will “return” to that location instead of `main()`!
Buffer overrun exploits

- In a real buffer exploit, the program would read the “payload” into the buffer which would contain a pointer to the payload.

```
STACK
POINTER
`-----> ____________
pushed| i | << stack frame for sum_integers()
by >| sum |
sum_int| buffer | local buffer, contains 0
egers| 1 |

| pushed| z3 | << stack frame for main()
by >| z2 |
main >| z1 |

| pushed| return address |
by >| argv |
caller| argc |
```

- `buf[134]` (buf has 64-bit longs, z2 and z3 are 32-bit ints)
- (on my laptop; your results may vary!)
Buffer overrun exploits

• In a real buffer exploit, the program would read the “payload” into the buffer, which would also overwrite the return address with a pointer to the payload

• Generally, the attacker must construct the payload in advance, so they need to know where the buffer is stored in memory
  – but that location may be intentionally constant (e.g. very low-level code may not use standard memory allocation)
  – or unintentionally almost-constant (basic server processes that are always run in the same order during the boot process)
How you could run your program on the entire Internet...

• We just missed the 30th anniversary of the Morris worm (2 Nov. 1988)
• Supposedly written “to gauge the size of the Internet”*
• Effects were amplified by a simple randomized algorithm:
  – usually, when you’re trying to cover a graph, you don’t try to cover nodes you already covered
  – so ask the OS if the worm is already running
  – Morris figured that sysadmins would patch their systems to always claim the worm was running

* 1988 was long before we stopped capitalizing “internet”
How you could run your program on the entire Internet...

• Effects were amplified by a simple randomized algorithm:
  – so, instead of not replicating if a copy seemed to already be running on the “new” host, he programmed the worm to replicate anyway with probability 1/7
  – the worm attempted to spread often enough that “1/7th” turned out to be “a lot”
How you could run your program on the entire Internet...

- Morris was convicted of violating the US *Computer Fraud and Abuse Act*
- DARPA created the “Computer Emergency Response Team” (CERT)
- The extent of the worm’s spread is unclear (maybe 10% of 60,000 machines on the Internet)

How you could run your program on the entire Internet...

• The main part of the worm could only infect the DEC VAX and the Sun-3, but these were popular systems at the time

• An important “attack vector” used was a vulnerability in the Unix `finger`d program...
How you could run your program on the entire Internet...

• The main part of the worm could only infect the DEC VAX and the Sun-3, but these were popular systems at the time

• An important “attack vector” used was a vulnerability in the Unix fingerd program...

• ...a buffer overrun!
  – fingerd used a 512-byte buffer (impressively large, since Unix usernames are limited to 8 characters) and called gets()
  – The Morris worm sent fingerd 536 bytes
SYNOPSIS

```
#include <stdio.h>

char *
fgets(char * restrict str, int size, FILE * restrict stream);

char *
gets(char *str);
```

...

SECURITY CONSIDERATIONS

The gets() function cannot be used securely. Because of its lack of bounds checking, and the inability for the calling program to reliably determine the length of the next incoming line, the use of this function enables malicious users to arbitrarily change a running program's functionality through a buffer overflow attack. It is strongly suggested that the fgets() function be used in all cases. (See the FSA.)

SEE ALSO

feof(3), ferror(3), fgetln(3), fgetws(3), getline(3)

STANDARDS

The functions fgets() and gets() conform to ISO/IEC 9899:1999 (``ISO C99'').
How you could run your program on the entire Internet...

• Security wasn’t taken that seriously
  – around 1992, all the servers at a certain department at a mid-size US university were broken into (possibly by reading unencrypted passwords in transit)
  – the sysadmin’s response (via the servers’ “welcome” message) was “this isn’t some paranoid high security operation, no hacker points for breaking in here”
How you could run your program on the entire Internet...

• But...so what if the worm could take over fingerd? It’s just a program, and Unix was always intended as a multi-user OS that enforced user privileges

• Wouldn’t you need to be a root program to do any real damage?
How you could run your program on the entire Internet...

fingerd ran as root
It gets worse

• In addition to showing you when (and from what remote host) a user had logged in, fingerd would display the contents of the user’s “.plan” file (sometimes a weekly schedule, so you’d know when the user might be free for a meeting)

• https://www.w3.org/History/1993/WWW/AccessAuthorization/FingerHole.html

“On some systems, the finger daemon, fingerd, was run under user-id zero (root). In this case a user could make his .plan file just to be a link to a read-protected file. ...”
It gets worse

• Aside: finger presented privacy concerns, but it was possible to turn the tables by making the .plan file be a “named pipe”, and running a program that did I/O on the pipe

• The program would write to a log file whenever the .plan file was read

• There was no direct way to know who was contacting the finger daemon, but you could run netstat and look for connections on port 79...
Friday Nov. 2 ended here

- a “palate cleanser”:
Principle of Least Privilege

• Wikipedia:
  – “every module...must be able to access only the information and resources that are necessary for its legitimate purpose.”

• ‘fingerd’ didn’t really need to be root
Principle of Least Privilege

- Unix systems started having a user called “nobody”, and having programs like fingerd run as that user

```
$ finger nobody
Login: nobody                     Name: Unprivileged User
Directory: /var/empty             Shell: /usr/bin/false
Never logged in.
No Mail.
No Plan.
```
Address Randomization

- Buffer overruns generally need to predict a **specific** stack pointer location.

- One way to foil attackers (or rather, reduce the damage to a crash rather than a zombie army) is **address space layout randomization**.

- This uses (pseudo)random numbers to make it much, much harder to guess where the stack will be.
Address Randomization

- [https://en.wikipedia.org/wiki/Address_space_layout_randomization](https://en.wikipedia.org/wiki/Address_space_layout_randomization)
  - Android: 2011: address layout,
    2015: order in which libraries are loaded
  - iOS 4.3: 2011
  - Linux: 2003: user processes, 2014: kernel location
  - Windows: 2007: some user processes
  - macOS: 2007: system libraries,
    2011: all user processes,
    2012: kernel location
Address Randomization

• Randomization can become less effective if the computer runs out of memory
  – there are fewer possible places to “randomly” put the process, increasing the chance an attacker’s guess will be correct
Walking through the open front door

- One way to “break in” to a house is by walking through an open front door
- The original Mac OS didn’t have a locked front door
- Arguably it didn’t even have a door
The Mac OS resource fork

• The Mac OS (1980s–’90s vintage) had a strange feature called a “resource fork”

• Files stored ordinary data (such as text) but every file also had a resource fork, which was a cute little database, the kind you might design if it was 1982 and you only had 128K of RAM
The Mac OS resource fork

• The resource fork contained key-value pairs, where a key was itself a pair \((TYPE, ID)\), with a 32-bit \(TYPE\) and a 16-bit \(ID\)

• The type told you what kind of resource it was

• For example, ‘ICON’ was the type of icons (black and white, 32 x 32) and ‘CODE’ was the type of application code

• Some kinds of code had their own resource type, such as ‘WDEF’ for window objects (ominous music plays)
The Mac OS resource fork

• Both the OS and user applications loaded resources by calling `GetResource(typ, id)`
• `GetResource` would look for that key in all the open resource forks, including open documents, open applications, and the OS itself, in that order
The Mac OS resource fork

• Applications could define icons for themselves, but also for their document types

MacPaint application

MacPaint document
The Mac OS resource fork

• Apple wanted to display correct icons that matched the application, even when the application hadn’t been installed on the hard disk

MacPaint application

Generic document icon; not good, doesn’t tell the user what application it’s for
The Mac OS resource fork

• Apple wanted to display correct icons that matched the application, even when the application hadn’t yet been installed on the hard disk

• The icon resources were stored in the “Desktop file”, an invisible file on every volume, including floppies

• When a disk was inserted, the Finder would open the Desktop file’s resource fork
The Mac OS resource fork

• Then, whenever the Finder needed to display a file icon, it just had to call `GetResource` and if any open Desktop file contained the icon, it would find the icon and display it
The Mac OS resource fork

• The Mac OS really liked resource forks, and the code to draw windows had its own resource type, WDEF (Window DEFINition)

• You could choose different window styles by specifying different WDEF ID numbers, but most people used WDEF #0

• You could create your own window styles by writing your own WDEF resource
The Mac OS resource fork

• The Desktop file was supposed to contain icons (and “bundles”, type BNDL, listing the ID numbers of different applications’ icons)

• However, resource forks were generic databases, so you could put any kind of resource in the Desktop file...
The Mac OS resource fork

• The Desktop file was supposed to contain icons (and “bundles”, type BNDL, listing the ID numbers of different applications’ icons)

• However, resource forks were generic databases, so you could put any kind of resource in the Desktop file...
  – including a WDEF resource
    • whose ID was 0, the same as the default system WDEF
The Mac OS resource fork

• Since the Desktop file was opened after the Finder started, attempting to draw a window would use the WDEF resource in the Desktop file

• You could get the Mac to run your virus code, basically, just by having your virus say, “Hi! I am in your general vicinity!”
The Mac OS resource fork

• Whoever wrote the WDEF virus did a really bad job, so in addition to copying itself to your hard disk’s Desktop file (and to any uninfected floppies you inserted), it crashed your computer a lot

• Upside: it helped users notice that something was wrong with their computer
The Mac OS resource fork

- WDEF was followed by CDEF, which had the same infection strategy
- Countermeasures were developed, including a free program Disinfectant
The Mac OS resource fork

• “virus.wikidot.com/wdef” says that “The [WDEF] virus is triggered when the Desktop file is executed”

• This is true, because the “resource chain” arrangement meant that opening any file could easily lead to executing code

• Supposedly, Freehand (a then-popular drawing program) and Excel were infected (you would open the shrink-wrapped package and the floppy inside would infect you)
And so on...

• The ’90s were also a great time for macro viruses that spread via Word documents, Excel documents, etc.

it feels like the worst viruses of the 90s weren't from elaborate attacks but just like really obviously (now) bad programming. Tons of executable stuff where it shouldn't have been.

―Kristopher Micinski
What do these have in common?

• **Remote code injection** is a key component of many attacks
• A buffer overrun tricks a program into executing the attacker’s code, instead of the program’s code
• WDEF “tricked” the early Mac OS into executing the virus’s code (barely a trick, because the Mac OS was happy to execute any code in sight)
• Macro viruses took advantage of programs like Word automatically running macros found in Word documents
What do these have in common?

• Like WDEF/CDEF, macro viruses declined in the 1990s as soon as the vendor of the targeted software reconsidered the approach
  – Apple stopped using resource forks for the Desktop file, making a new file format specifically for the intended contents of the Desktop file
    • similar to null pointers—you often don’t ever want nulls for some kinds of data, and Apple never intended the Desktop file to contain executable code
  – Microsoft stopped automatically running macro code
Technical Exploit → User Exploit

• Macro viruses depended on “persuading” Word/Excel to automatically run them

• Once Word and Excel stopped doing that, the problem was solved...right?
Technical Exploit → User Exploit

• Macro viruses depended on “persuading” Word/Excel to automatically run them
• Alas, technical exploits often have an equivalent user exploit
  – Can you guess what user exploit a Word macro virus could use instead?
Technical Exploit → User Exploit


• Macro viruses peaked in 1999, then declined

• Resurgence around 2013

• Just have to persuade the user to manually enable the macros!
Heartbleed

- “both a name and a logo”
- “catastrophic”
- ~17% of HTTPS servers “were believed to be vulnerable”
- **Not** a buffer overrun...
Heartbleed: Background

• Secure HTTP (HTTPS) was introduced in the late ’90s

• Got a boost in 2013 after Edward Snowden leaked information about government surveillance activities
  – it was already well-known among certain groups that the US NSA was spying on a massive scale
  – back around 1990, some people put “hot” keywords in their email and Usenet signatures, known as “food for the NSA’s line-eater”
Heartbleed: Timeline

• April 1, 2014: Reported privately
• April 7, 2014: Disclosure to the public
• April 8, 2014: 900 SINs stolen via CRA website
  – “RCMP...charged a computer science student”
• “Anti-malware researchers also exploited Heartbleed to their own advantage in order to access secret forums used by cybercriminals.”
Heartbleed: Cartoon Version

- https://xkcd.com/1354/
Heartbleed: why?

- What is this “server, are you still there?”
- Secure HTTP requires “negotiating a connection”, which takes time
- Can avoid by continuously transferring data, which also takes time and wastes bandwidth
- So in 2012, the RFC 6520 protocol was created to allow browsers to keep the connection “alive” without full negotiation
A buffer exploit, but not an overrun

- Heartbleed is not an overrun; no malicious code is injected
- Instead, it is an **information leak**
  - a user has no right to see other users’ information that was left over in the buffer
A buffer exploit, but not an overrun

- Heartbleed is a relatively blatant information leak
- Less blatant leaks include:
  - timing (as mentioned near the beginning of these slides)
Stopping Information Leaks

• **Information-flow type systems** can catch information leaks
  – If the system passes the type checker, the program doesn’t leak high-security information to low-security modules
  – Which leads to an interesting problem...
Stopping Information Leaks

• **Information-flow type systems** can catch information leaks
  
  – Remember our password example?
  
  – A low-security process wants to log in to an account
    
    • this process **must** be low-security, otherwise, it would be allowed to read the password file directly!
  
  – So it needs to pass the characters typed by the user to a high-security system function
Stopping Information Leaks

• **Information-flow type systems** can catch information leaks
  
  – Remember our password example?
  
  – A low-security process wants to log in to an account
    
    • this process **must** be low-security, otherwise, it would be allowed to read the password file directly!
  
  – So it needs to pass the characters typed by the user to a high-security system function that returns **true** if the password matches, and **false** otherwise
Stopping Information Leaks

• **Information-flow type systems** can catch information leaks
  – But that leaks information about the stored password, so the type checker rejects this!
  – So we need *quantitative* or *probabilistic* approaches that say, “yes, high-security information is leaked, but the amount is too small to worry about” (cf. information entropy)
Conclusion

• Like other aspects of software quality, **security is not accidental**; it must be considered throughout the design of a system

• Many aspects of security are not technical, so they don’t have purely technical solutions

• Technical “attack vectors” include
  – buffer overruns
  – information leaks
Next Time

• We’ll look at how to protect against buffer overruns more systematically
• And how the choice of programming language influences software quality, more generally