The State of Practice in Software Security and its Foreseeable Evolution

Class 01, Monday 09-20-2015, 5:10–7:00, WWH–312

Patrick Cousot
p cousot@cs.nyu.edu  cs.nyu.edu/~pcousot

TODO

• On individual sheets of paper provide your FIRST NAME LAST NAME EMAIL all written very clearly (e.g. uppercase)

• You will receive at this EMAIL a login and a password to access the course slides, located at URL:
http://cs.nyu.edu/~pcousot/courses/fall15/CSCI-GA.3033-011/slides/index.html

All communications for this course will be done through this EMAIL

• Office hours are 4:00PM – 4:55PM, in my office: CIWW 405 by appointment, email to pcousot@cs.nyu.edu

SLIDES

• The slides will be made available on the class web site (after each class):
http://cs.nyu.edu/~pcousot/courses/fall15/CSCI-GA.3033-011/slides/index.html

• The slides serve a double purpose
  • Support of oral presentation (should be very light)
  • Support for reading after the class (should be self-contained and comprehensive)
    ➔ a difficult compromise

• Reading and understanding the slides is part of the mandatory homework

Level of the course

• This is a PhD level course opened to (the best) master students

• Some basic knowledge is presupposed and not provided in the class

• Includes the ability to read recent, up to date, and possibly difficult research articles (PhD students have to read research papers)

• Not a hacking course (no programming required)

• Emphasis is on the methodology (thinking required)
Warning

- The first class is completely informal, if not anecdotic, which unfortunately is the state of the art
- The following classes are more formal, which will be the state of the art in a few years
- Have a look at the prerequisites!

State of Practice

- The software industry, and in particular the software security industry, completely fails on security
- Proved daily by new attacks

The State of Practice in Software Security

Example 1 of attack
Wall Street’s Exposure to Hacking (NYT)


- “Five people were accused of hacking the Nasdaq stock exchange for two years, showing the financial system’s vulnerability to cybercrime.”

- “Mr. Kalinin, first cracked Nasdaq’s systems in late 2007 using so-called SQL injections.”

What is SQL injection?

- SQL: language to manipulate data held in database.

- Example of code subject to injections:

  - check if userName is registered in database

  ```plaintext
  statement = "SELECT * FROM users WHERE name = '" + userName + '";"
  ```

  - a malicious userName answer: ' or '1'='1

  - the executed statement is

  ```plaintext
  SELECT * FROM users WHERE name = '" OR '1'='1';
  ```

  which is true → access authorized!

Examples of SQL injection attacks

- August 2014, Milwaukee-based computer security company Hold Security disclosed that it uncovered a theft of confidential information from nearly 420,000 websites through SQL injections.

- June 27, 2013, hacker group “RedHack” breached Istanbul Administration Site.

- October 1, 2012, a hacker group published the personal records of students, faculty, employees, and alumni from 53 universities including Harvard, Princeton, Stanford, Cornell, Johns Hopkins, and the University of Zurich on pastebin.com.

- July 2012 a hacker group was reported to have stolen 450,000 login credentials from Yahoo!

- May 2012, the website for Wurm Online, a massively multiplayer online game, was shut down from an SQL injection while the site was being updated.

- June 2011, Public Broadcasting Service was hacked, mostly likely through use of SQL

- June 1, 2011, “hacktivists” of the group LulzSec were accused of using SQLi to steal coupons, download keys, and passwords that were stored in plaintext on Sony’s website, accessing the personal information of a million users

- Over a period of 4 hours on April 27, 2011, an automated SQL injection attack occurred on Broadband Reports website that was able to extract 8% of the username/password pairs:

- On April 11, 2011, Barracuda Networks was compromised using an SQL injection flaw. Email addresses and usernames of employees were among the information obtained.

  see many more on [http://en.wikipedia.org/wiki/SQL_injection](http://en.wikipedia.org/wiki/SQL_injection) + all the ones that remain undiscovered

Preventing SQL injection errors

- Prevention is very easy:

  - All inputs should be sanitized (e.g. by checking that it does not contain any special character)

  - Not sanitizing input should be considered as a malpractice / professional error

    but it isn’t!

  - Prevention is very easy, can be automatized by static analysis
Example II of attack

AtomicReferenceArray vulnerability on Java

- Multiple versions of Oracle Java Runtime Environment (JRE) contain a vulnerability that could allow an unauthenticated, remote attacker to execute arbitrary code on a targeted system.

- The vulnerability exists because the *AtomicReferenceArray* class could perform deserialization of an object array in an unsafe manner. Due to this unsafe deserialization, an unauthenticated, remote attacker could bypass the JRE sandbox mechanism and execute arbitrary code on the system outside the sandbox protections.

- An unauthenticated, remote attacker could exploit this vulnerability by persuading a user to visit a malicious web page. The processing of malicious Java content in the web page could trigger a logical flaw that the attacker could use to bypass security sandbox mechanisms and execute arbitrary code with the privileges of the user.

The bug at the origin of the error


  “This vulnerability exists because the *AtomicReferenceArray* class is not checking properly whether the array is an appropriate object type.”

- The bug should be considered as a malpractice / professional error

  but it isn’t!

- Prevention is very easy, can be automatized by static analysis!
History of security attacks

- See a Timeline of computer security hacker history at
- The same programming errors are repeated again and again and again
- This state of practice should be considered as a malpractice / professional error but it isn’t!
- Prevention is very easy, can be automatized by static analysis!

Security web site

- A list of websites devoted to security is provided at
  http://www.lib.iup.edu/comscisec/secweb.htm
- A list of vulnerabilities is provided at URL
  https://www.owasp.org/index.php/Category:Vulnerability
  http://www.sans.org/top25-software-errors/

Who is held responsible for bad designs?

- The attacker, of course
- What about the programmer?
  - programmers are not held responsible for the malpractive / professional errors (the right to error)
- What about the designers of the SQL/Java languages?
  - no responsibility for bad and unsafe language design
State of practice in manufacturing

- Automobile/aviation/train manufacturers
- Held responsible for correct design
- Errors are investigated
- Example: Toyota's Electronic Throttle Control System: the National Highway Traffic Safety Administration (NHTSA) commissioned the National Academy of Sciences (NAS) for an independent review (entitled "The Safety Promise and Challenges of Automotive Electronics: Insights from Unintended Acceleration").

(\textsuperscript{(*)}) \url{http://www.nap.edu/nap-cgi/report.cgi?record_id=13342&tipo=pdfxsum}

State of practice in manufacturing

- On Thursday October 24, 2013, an Oklahoma court ruled against Toyota in a case of unintended acceleration that lead to the death of one the occupants. Central to the trial was the Engine Control Module's (ECM) firmware
- Michael Barr, CTO and co-founder of Barr Group, and his colleagues, illuminate a shameful example of software design and development:
  - Toyota's electronic throttle control system (ETCS) source code is of unreasonable quality.
  - Toyota's source code is defective and contains bugs, including bugs that can cause unintended acceleration (UA).
  - Code-quality metrics predict presence of additional bugs.
  - Toyota's fail safes are defective and inadequate (referring to them as a "house of cards" safety architecture).
  - Misbehaviors of Toyota's ETCS are a cause of UA.
- see: \url{http://www.edn.com/design/automotive/4423428/Toyota-s-killer-firmware--Bad-design-and-its-consequences}

State of practice in manufacturing

- Example of identified error: Stack overflow
  - Toyota claimed only 41% of the allocated stack space was being used.
  - Barr's investigation showed that 94% was closer to the truth.
  - Although Toyota had performed a stack analysis, Barr concluded the automaker had completely botched it. Toyota missed some of the calls made via pointer, missed stack usage by library and assembly functions (about 350 in total), and missed RTOS use during task switching. They also failed to perform run-time stack monitoring.

(\textsuperscript{(*)}) \url{http://www.nap.edu/nap-cgi/report.cgi?record_id=13342&tipo=pdfxsum}

State of practice in manufacturing

- From the NAS report (\textsuperscript{(*)}):
  - Finding 3.2: Testing, analysis, modeling, and simulation are used by automotive manufacturers to verify that their electronics systems, the large majority of which are provided by suppliers, have met all internal specifications and regulatory requirements, including those relevant to safety performance.
  - NHTSA's Office of Defects Investigation (ODI) can anticipate ... recalls involving software reprogramming and other fixes to the hardware of electronics systems.
State of practice in manufacturing

- **Finding 4.6:** The Federal Aviation Administration (FAA) exercises far greater over-sight of the verification and validation of designs and their implementation.
- Rapid changes in dealing with the safety of software used in critical airborne systems.

State of practice in computer science

**MICROSOFT SOFTWARE LICENSE TERMS:**

- **DISCLAIMER OF WARRANTY.** The software is licensed "as-is." You bear the risk of using it. Microsoft gives no express warranties, guarantees or conditions. Microsoft excludes the implied warranties of merchantability, fitness for a particular purpose and non-infringement.
- **LIMITATION ON AND EXCLUSION OF REMEDIES AND DAMAGES.** You can recover from Microsoft and its suppliers only direct damages up to U.S. $5.00. You cannot recover any other damages, including consequential, lost profits, special, indirect or incidental damages.

The reason for this unfortunate situation?

- Software companies win all their trials in courts because of a simple argument:
  - Software bugs are part of the state of practice
  - Judges cannot request more of a professional than to be up to date with the state of practice of their field

State of the Art

- The **state of the art** is much more advanced than the state of practice
- **Automatic tools** can detect (and sometimes even correct) most of the software bugs linked to security
- A professional not using an existing tool to detect vulnerabilities later causing damages **might be held responsible** for not being “state of the art”:
  - Impossible to argue for the **right to error**
  - Possible to argue for **incompetence by ignorance** of the existence of relevant tools
State of art in manufacturing

- Example: state of the art for stack overflow
- There are completely automatic tools to prove the absence of stack overflow in software (for all inputs and task executions)
- Example: StackAnalyzer (http://www.absint.com/stackanalyzer/index.htm)

What is changing?

- “With the increasing incidence of hacking, denial-of-service attacks, penetration, malware infection, and botnet creation, one might be tempted to argue that software developers should bear more responsibility for the behavior of their software.” (*)
- “We are so dependent on an increasing number of programs, large and small, it is difficult to believe the software profession will escape some kind of deep accountability in the future. We may avoid this in the near term, but I am not so sure in the long term.” (*)

(*) Vinton G. Cerf: ‘But officer, I was only programming at 100 lines per hour!’, Commun. ACM 56(7): 7 (2013)

State of practice ≠ State of the art

- Teaching the state of practice is easy and enjoyable (for hackers)
- But, this is teaching the state of malpractice
- Teaching the state of the art is an ethical requirement
- But, this is much more difficult and much more demanding for students:
  - Formal methods
  - Mathematics
  - Proofs

Consequences for the course content
Background

“factual and circumstantial information that is essential to full understanding of a particular problem or situation” (Merriam Webster)

Required background

- **Computer Science:**
  - Programming languages (good practice of programming in any high-level programming language)
  - Principles of operating systems
  - Principles of compilation
- **Mathematics:**
  - Logic, set theory, order theory
  - A basic knowledge in formal methods.

Homework

- Acquiring the prerequisites is necessary and important
- Studing the slides is mandatory
- No programming project
- Exercices are provided as facultative homework
  - No solution given (researchers have to find their own solutions to their problems)
  - But, you can discuss your solution with
    - classmates (researchers present their solutions to colleagues for discussion and evaluation)
    - the instructor (during office hours)
The first class

Introduction

- Introduce some of the topics studied in the course:
  - Buffer overrun analysis (static analysis by abstract interpretation)
  - Confidentiality, integrity, availability, declassification and erasure
  - Tainting, dependence analysis, language-based information-flow security
  - Access control, stack inspection
  - Trust management, in-lined reference monitors
  - Static analysis techniques and type systems for enforcing security policies
  - Proof-carrying code, certifying and certified compilation, and security-preserving compilation
  - Verification of cryptographic protocols
  - Global computing, network, Internet, and web vulnerabilities and defences

- No formal content (contrary to following classes)

Content

- Programming failures
- Buffer overflows
- Race conditions
- Permissions and access control
- Poor randomness
- Confidentiality leaks
- Program design in security: guidelines

Programming and security

- The relationship between programming and security may be viewed in at least a couple of ways.
  - Programming Securely
  
  To develop code in a secure manner so that the code itself is not a vulnerability that can be exploited by an attacker

  - Programming Security
  
  To develop code for security-specific functions such as encryption, digital signatures, firewalls, etc.

The second may be required for the first. In this class, we consider Programming Securely.
Vulnerabilities at CERT/CC

- The CERT Coordination Center
  
  [http://www.cert.org](http://www.cert.org)

- at Carnegie Mellon University is a reporting centre for Internet security problems.

- CERT/CC provides technical advices, recommended responses, identifying trends.

Statistics available at

[http://www.cert.org/stats/cert_stats.html#vuls](http://www.cert.org/stats/cert_stats.html#vuls)
Categories of Vulnerabilities

- The main vulnerabilities where buffer overflows, up to 50% in 2005
- Considerable progress has been made on buffer overflows, in particular thanks to static code analysis
- More recently, with the rise of web applications written in higher-level languages, it has been taken over by SQL injection and OS command injection.

Most dangerous software errors


<table>
<thead>
<tr>
<th>Rank</th>
<th>Score</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>93.8</td>
<td>CWE-89</td>
<td>Improper Neutralization of Special Elements used in an SQL Command</td>
</tr>
<tr>
<td>[2]</td>
<td>83.3</td>
<td>CWE-78</td>
<td>Improper Neutralization of Special Elements used in an OS Command</td>
</tr>
<tr>
<td>[3]</td>
<td>79.0</td>
<td>CWE-120</td>
<td>Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')</td>
</tr>
<tr>
<td>[4]</td>
<td>77.7</td>
<td>CWE-79</td>
<td>Improper Neutralization of Input During Web Page Generation</td>
</tr>
<tr>
<td>[6]</td>
<td>76.8</td>
<td>CWE-862</td>
<td>Missing Authorization</td>
</tr>
<tr>
<td>[7]</td>
<td>75.0</td>
<td>CWE-798</td>
<td>Use of Hard-coded Credentials</td>
</tr>
<tr>
<td>[8]</td>
<td>75.0</td>
<td>CWE-311</td>
<td>Missing Encryption of Sensitive Data</td>
</tr>
<tr>
<td>[9]</td>
<td>74.0</td>
<td>CWE-434</td>
<td>Unrestricted Upload of File with Dangerous Type</td>
</tr>
<tr>
<td>[10]</td>
<td>73.8</td>
<td>CWE-807</td>
<td>Reliance on Untrusted Inputs In a Security Decision</td>
</tr>
<tr>
<td>[11]</td>
<td>73.1</td>
<td>CWE-250</td>
<td>Execution with Unnecessary Privileges</td>
</tr>
<tr>
<td>[12]</td>
<td>70.1</td>
<td>CWE-352</td>
<td>Cross-Site Request Forgery (CSRF)</td>
</tr>
</tbody>
</table>

Overflows
Examples of software with buffer overflows

- at, AutoSofts RTS inventory control system, bash, bind, catdoc, chkey, cidentd, Cisco 7xx routers, Cisco IOS, compress, cron, CSM, cxterm, dbadmin, DCE secd, deliver, df, dig, dslip, dtaction, dtterm, eeprom, eject, elm, elves, fconfig, fdformat, ffconfig, filter (part of elm-2.4), FreeBSD's crto.c, gethostbyname(), getopt(), host, imapd, ipop3d, iwsh, jidentd, junkbuster, klogd, lha, libauth, libDxSvc, libX111, libXt and further X11R6 libraries, login, login/scheme, Lotus Notes, lpd, lpr, lquerylv, metamil, MH-6.83, Minicom, modstat, mount, mount/umount, mountd, mpg123, mscreen, MSIE, mutt, Netscape (all versions), Netscape Mail, newgrp, ...

Origin of buffer overflows/overruns

Programmers are often careless about checking the size of arguments, storing them into fixed size buffers using functions which don’t check for overflow.

Classically, the problem is with C-style strings, implemented as arbitrary length null-terminated sequences of bytes.

Standard C library functions like strcpy() do not check bounds when copying from source to destination. If the destination buffer is too small, the string will overflow and corrupt other data.

Examples of software with buffer overflows

- .../... NIS+, nispasswd, Novell Groupwise, nslookup,.opengroup, ordist, ospf_monitor, Outlook Express, passwd, pcnfs, permissions, pine, ping, portmir, Proxy, ps, pset, rcp, rdls, rdist, Real Player, rlogin, rpc.nisd, rsh, Samba server, sdtcm_convert, sendmail for Windows NT, sendmail, SerialPOP, SGI (an entire range of programs), SLMail, smbclient, smbmount, socks5, splitvt, statd, suidperl, SuperProbe, syslog, talkd, telnetd, term, termcap, tgetent(), the IMail POP3 server for NT, the Linux dynamic linker, the NLS library, the query CGI script, the Windows War and Serv-U FTP daemon, the Yapp conferencing server, Tooltalk, traceroute, ttbdserver, ufstore, unfsd, WebCam32, WebSite 1.1, Windows95/NT NTFTP client (multiple problems), wm, Word, writesrv, wwwwcount, Xaw library, xdat, xlock, Xprt, xscreensaver, Xserver, xterm, yppassword, zvg, zpop, etc.

Example 1

- The following code asks the user to enter their last name and then attempts to store the value entered in the last_name array

```
Example Language: C

char last_name[20];
printf ("Enter your last name: ");
scanf("%s", last_name);
```

- The problem with the code above is that it does not restrict or limit the size of the name entered by the user. If the user enters "Very_very_long_last_name" which is 24 characters long, then a buffer overflow will occur since the array can only hold 20 characters total.
Example 2

- The following code attempts to create a local copy of a buffer to perform some manipulations to the data.

```c
Example Language: C

void manipulate_string(char* string){
    char buf[24];
    strcpy(buf, string);
    ...
}
```

- The programmer does not ensure that the size of the data pointed to by string will fit in the local buffer and blindly copies the data with the potentially dangerous `strcpy()` function. This may result in a buffer overflow condition if an attacker can influence the contents of the string parameter.

**Bad code!**

Example 3

- The excerpt below calls the `gets()` function in C, which is inherently unsafe.

```c
Example Language: C

char buf[24];
printf("Please enter your name and press <Enter>\n");
gets(buf);
...
```

- The programmer has used the function `gets()` which is inherently unsafe because it blindly copies all input from STDIN to the buffer without restricting how much is copied. This allows the user to provide a string that is larger than the buffer size, resulting in an overflow condition.

**Bad code!**

Example 4

- In the following example, a server accepts connections from a client and processes the client request.

- After accepting a client connection, the program will obtain client information using the `gethostbyaddr` method, copy the hostname of the client that connected to a local variable and output the hostname of the client to a log file.

```c
... struct hostent *clienthp;
    char hostname[MAX_LEN];
    // create server socket, bind to server address and listen on socket ...
    // accept client connections and process requests
    int count = 0;
    for (count = 0; count < MAX_CONNECTIONS; count++) {
        int clientlen = sizeof(struct sockaddr_in);
        int clientsocket = accept(serversocket, (struct sockaddr *)&clientaddr, &clientlen);
        if (clientsocket >= 0) {
            clienthp = gethostbyaddr((char *)&clientaddr.sin_addr.s_addr,
                sizeof(clientaddr.sin_addr.s_addr), AF_INET);
            strcpy(hostname, clienthp->h_name);
            logOutput("Accepted client connection from host ", hostname);
            // process client request ...
            close(clientsocket);
        }
        close(serversocket);
    }
```

**Bad code!**
Example 4 (cont’d)

- However, the hostname of the client that connected may be longer than the allocated size for the local hostname variable.
- This will result in a buffer overflow when copying the client hostname to the local variable using the `strcpy` method.

Errors due to buffer overflows

- Overflows can corrupt other pieces of the program data and cause security bugs, or even execution of arbitrary code...
- **Availability attack I**: Buffer overflows generally lead to crashes.
- **Availability attack II**: Other attacks leading to lack of availability are possible, including putting the program into an infinite loop.

Attacks using buffer overflows

- **Attacks on the stack**: exploit a program that uses a stack allocated buffer, by using a specially constructed longer-than-expected argument. See [http://en.wikipedia.org/wiki/Stack_buffer_overflow](http://en.wikipedia.org/wiki/Stack_buffer_overflow)
- At the code level, stack-based and heap-based overflows do not differ significantly, so there usually is not a need to distinguish them.
- From the attacker perspective, they can be quite different, since different techniques are required to exploit them.

Example: smashing the stack

The malicious argument of a function over-writes all of the space allocated for the buffer, all the way to the return address location.

This return address location is altered to point back into the stack, somewhere in a “landing pad” of NOPs before the attack code (the “egg”).

Typically the attack code executes a shell.
**Detection methods**

- **Manuel analysis**: code audit to find vulnerable code, safe libraries with bound checks
- Problems of human cost, lack of guaranteed coverage, etc

---

**Detection methods (cont’d)**

- **Automated dynamic analysis**:
  - *buffer overrun runtime checks* inserted by the compiler (may be too costly and provide no alternative in case of error but raising an exception)
- Disabling stack or data execution:
  - Execute Disable Bit (NX) now added to Intel and AMD CPUs
  - Needs operating system support (added in Windows XP SP2, Linux 2.6.8)

---

**Detection methods (cont’d)**

- **Automatic static analysis** (done at compile/test time so no cost at runtime)

---

**Race-conditions**

---

*John Wilander, Mariam Kamkar: A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention, NDSS 2003*
Race conditions

- A race condition or race hazard is a flaw in an electronic system or process whereby the output or result of the process is unexpectedly and critically dependent on the sequence or timing of other events.
- The term originates with the idea of two signals racing each other to influence the output first.
- Race conditions can occur in electronics systems, especially logic circuits, and in computer software, especially multithreaded or distributed programs.

Example

Unix `mkdir` used to work in two stages:

1. Make new directory
2. Change ownership

Attack:

- suspend `mkdir` process between 1 and 2,
- replace new directory with a link to a confidential file, e.g., `/etc/passwd`.
- Resume process; it then changes permissions on the critical file instead of the new directory.

Detecting and preventing race conditions

Race conditions can be hard to detect, because they arise due to asynchronous processing (e.g. multiple threads) and may seldom/never occur during ordinary use.

Likely to be a growing problem.

General approaches to fixing:

- synchronization: use locks in multi-threaded programming (can be costly due to waiting times);
Access control

• Who can do what on your computing system?

• The 1970s: one computer-one programmer/user, machine-man communication via card/tape readers, printers, teletypes.

  ➞ no protection needed at the operating system level (prevent physical access to the machine)

Access control (cont’d)

• The 1980s: one computer-many programmers/users, possibility of sharing computing time, devices, files.

  ➞ protection required at the operating system level (control access of users to shared resources)


Access control (cont’d)

• The 1990s: one computer-many programmers-many users accessing services via internet

  ➞ protection required both at the operating system and the application/service level (control access of users to private resources via privileges when interacting with services)

  ➞ see basic access control concepts at http://en.wikipedia.org/wiki/Access_control

Permission vulnerabilities

Many exploits have taken advantage of failure to follow the principle of least privilege (*), see https://en.wikipedia.org/wiki/Principle_of_least_privilege

Poor programming or inflexible OS permissions structures can lead to programs and users that are given more privileges than they need.

Typical pattern of attacking a system is using escalation of privilege (see http://en.wikipedia.org/wiki/Privilege_escalation):

Access control (cont’d)

• The 2000s: development of attacks, cyberattacks, cyberwarefare and counter-measures
  ➞ see http://en.wikipedia.org/wiki/Cyberwarfare
  ➞ see e.g. cross-site scripting at http://en.wikipedia.org/wiki/Cross-site_scripting

Access control (cont’d)

• The 2010s: research and development of cybersecurity at the machine, operating system, network/cloud and cyber-application levels
  ➞ see e.g. https://en.wikipedia.org/wiki/Computer_security
  ➞ this course is mainly oriented towards the security of cyber-applications

Granularity of security provision

• The hardware level has fine grained access controls.

• At higher levels, increasingly user-oriented security policies are implemented.

• Reliability of each level depends on levels below, and has increasingly complex implementations.
  • Hardware course
  • OS course
  • Network course
  • Software security course

Poor randomness
Random numbers

• On computers random numbers are pseudo-random and in fact
  • the sequence of draw completely predictable from the seed (see http://en.wikipedia.org/wiki/Random_number),
  • contrary to e.g. a lottery (see https://en.wikipedia.org/wiki/Lottery)
• Security applications require random numbers for various reasons, the most important of which is key generation (see http://en.wikipedia.org/wiki/Key_generation).
• Numerous exploits have taken advantage of the predictability of supposedly “random” numbers.

Pseudo-random numbers

• General strategy: true random seed + cryptographically strong PRNG (pseudorandom number generator, see https://en.wikipedia.org/wiki/Pseudorandom_number_generator) if more bits needed.
• Secure PRNG passes statistical randomness properties and has property that an attacker cannot guess the next value in the sequence based on some history of previous values.

Random seed

• How do we get the true random seed? Without a dedicated random source, we must rely on non-deterministic external environmental data.
  • Good choices: disk-head seek times, keystrokes, mouse movements, memory paging behaviour, network status, interrupt arrival times, random electrical noise (e.g. /dev/audio). Best to use several, combined with a hash (see http://www.ietf.org/rfc/rfc1750.txt).
  • Bad choices: system clock, Ethernet addresses or hardware serial numbers, network arrival packet timing or anything else that can be predicted or influenced by an adversary.

Random numbers in Linux

Unix’s random kernel device uses an “entropy pool” and estimates the number of “true” random bits in the pool (see https://en.wikipedia.org/wiki/dev/random).
Adding random data into pool recharges entropy; reading random bytes removes entropy.
Strong random device /dev/random can return no more bits than are in the pool.
Less secure device /dev/urandom returns unbounded amount of cryptographically strong numbers.
Confidentiality leaks

Security applications store sensitive data in data-structures held in memory.

Vulnerabilities:

- Other processes may be able to read memory
- Memory may be swapped to disk swap files
- Laptop BIOSes, OSes suspend-to-disk operation
- Data can be recovered from RAM after power down
- Journaling file systems, disk-caching make sanitization tricky
- Hard-disk data recovery can recover several generations of data
- TEMPEST radio frequency (RF) leakage from cables, monitors.

Some defences

- Touch memory regularly or OS calls to lock pages.
- Use custom swap file, zeroed after use.
- Blurred/anti-aliased fonts: reduce high-frequency RF
- Other defences beyond realm of software, see http://en.wikipedia.org/wiki/Tempest_(codename).

A vulnerability discovered in 2002

- Temporarily store a user's password in clear text.
- To minimize the lifetime of sensitive information, the password buffer is zeroed-out before returning to the heap:
  ```
  memset(password, '\0', len);
  free(password);
  ```
- A compiler that performs "dead-code" elimination will see that the program never uses the values written by the memset statement and will remove the statement, thereby leaving sensitive information exposed in the heap.
OS protection (access control)

From Salzer and Schroeder (1975), The Protection of Information in Computer Systems, CACM 17 (7), 1975, see http://www.cs.virginia.edu/~evans/cs551/saltzer/

1. **Economy of mechanism** — keep design simple and small as possible. Especially important in security because errors in design are not seen in normal use. Consider line-by-line code inspection.

2. **Fail-safe defaults** — base access decisions on permission rather than exclusion. Conservative design must argue why objects should be accessible, rather than why they should not.

3. **Complete mediation** — every access to every object must be authorized. This implies that a foolproof method of authentication is available.

4. **Open design** — the design should not be secret. Decouple protection mechanisms from protection keys; no security-by-obscurity.

5. **Privilege separation** — require two keys rather than one. Once the mechanism is locked, two distinct owners can be made responsible for the keys. Implementation of ADTs uses this idea.

6. **Least privilege** — every program and every user should operate using least privilege necessary for the job. Like military rule of “need to know”, see http://en.wikipedia.org/wiki/Principle_of_least_privilege.

7. **Least common mechanism** — minimize mechanisms common to more than one user; every shared mechanism is a potential information path.

8. **Psychological acceptability** — users should routinely, automatically use protection correctly; mechanisms should match their mental models.
Even more principles

- **Accountability** (log who does what, trace origin of principals)
- **Work factor** (comparison of cost of circumvention with the resources of an attacker, see [http://en.wikipedia.org/wiki/Cryptography](http://en.wikipedia.org/wiki/Cryptography))
- **Compromise recording** (make mechanisms tamper-evident).

Principles for secure programming

- General principles for **security programming** emerge from case history of security vulnerabilities.
- See for example
  - The CERT Coordination Center
    - [http://www.cert.org](http://www.cert.org)
  - The community-developed dictionary of software weakness types
    - [http://cwe.mitre.org](http://cwe.mitre.org)

Basic security programming principles

1. **Protect internal data and functions.** Use *language based access controls* (ADTs, visibility modifiers, modules).
2. **Handle impossible cases.** Today’s “impossible” cases may be quite likely in next week’s version. Introduce explicit errors (exceptions, assertions), do not assume these cannot occur.
3. **Use cryptography carefully.** Avoid predictable keys, small key spaces (choose cryptographic PRNGs and good seeds); be careful with key management (use secure locations, clear memory).
4. **Program defensively.** Beware data that comes from outside, and be aware of vulnerabilities introduced by relying on external programs. Try to minimise those vulnerabilities.
Application to C programming

1. Check all input arguments for validity. Since C is not strongly typed, the validity of types should be checked. Semantical checks should also be performed; e.g., if input is an executable file, should check that the file is indeed executable and user has execute permission for file.

2. Never use `scanf` (see [https://en.wikipedia.org/wiki/Uncontrolled_format_string](https://en.wikipedia.org/wiki/Uncontrolled_format_string)); use `fgetc` (similarly, avoid `printf`, etc.). In general, avoid routines which do not check buffer boundaries.

3. Perform bounds checking on every array index when in any doubt.

4. Check error return values. Essential because C doesn’t implement an exception mechanism.

5. Don’t keep secret information in memory of unprivileged programs; it may be possible to interrupt the program and cause it to dump core.

6. Consider logging UIDs, file accesses, etc.

7. Strip binaries (strings can reveal a lot!).

8. Etc, the list is far from complete!

Application to Unix programming

Be careful about relying on environment variables or other settings inherited from the environment (`umask`, etc.).

Use full pathnames for any filename, program or data. Use `chroot()` prisons to restrict access to a protected subdirectory.

Be very wary of the unix `system()` call (or similar `shell()`, `popen()`, `exec` family). It will execute whatever is passed.

Don’t use `chmod()`, `chown()`, `chgrp()`. Use `fchmod()`, `fchown()` instead, which use file descriptors instead of names, so do not involve separate opens (to avoid the race condition).

Take care with root permissions: beware of `setuid` programs, avoid setuid scripts, never open a file as root. If you need `setuid` root, give it up as soon as possible. Better to use ad-hoc usernames.

Conclusion and bibliography

• **Computer security** is a great research, development and applied subject of the 2010’s et certainly much beyond!

• This course on “Principles of Software Security” comes in complement to computer security issues considered in the hardware, operating system, network, and theory (cryptography) courses.

• An illustration of the use of formal methods (mathematically based techniques for the specification, development and verification of software and hardware systems).
Bibliography

- Secure Programming for Linux and Unix HOWTO — Creating Secure Software by David A. Wheeler, see Ch. 1, 2 and 5, freely available online http://www.dwheeler.com/secure-programs/

- See also:

Exercice (facultative homework)

- Find an input such that a program like the following one will, without any modification, print “I am not supposed to print anything here.” (the solution, if any, which consists in returning from the call inputT(T) after exit (0), is machine, compiler, linker, and system dependent, see e.g. http://www.tenouk.com/Bufferoverflowc/Bufferoverflow1-6.html).

```c
#include <stdio.h>
#include <stdlib.h>

void inputT(char t[])
{
    scanf("%s",t);
}

int main ()
{
    char T[1];
    inputT(T);
    exit (0);
    printf("\nI am not supposed to print anything here.\n");
}
```

Comment on the exercice

- An exercice for hackers!
- The difficulty is that it requires an intimate knowledge of machines, compilers, and languages
- The only one of that sort in the course =)
- Most students will fail on it
- Not important since the course is on prevention, not attack!

The End