Programming Securely I

Computer Security Lecture 11

David Aspinall

School of Informatics
University of Edinburgh

4th March 2014

Outline

- Programming failures
- Buffer overflows
- Race conditions
- Permissions and Access Control
- Poor randomness
- Confidentiality leaks
- Building in security: design and guidelines

Outline

Programming failures

Buffer overflows

Race conditions

Permissions and Access Control

Poor randomness

Confidentiality leaks

Building in security: design and 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.

Of course, the second may be required for the first.

In this lecture, we consider Programming Securely.

Choose security

2:22 pm PT, Tuesday, January 15, 2002.
*“. . . we’re in the process of training all our developers in the latest secure coding techniques. . . now, when we face a choice between adding features and resolving security issues, we need to **choose security**.”*

Choose security

2:22 pm PT, Tuesday, January 15, 2002.
*"...we're in the process of training all our developers in the latest secure coding techniques...now, when we face a choice between adding features and resolving security issues, we need to **choose security**."*

- The **penetrate and patch** approach to fixing security problems in mass market systems is badly flawed, e.g.
  - patches often do not get applied
  - patches often fix only symptoms, not cause
  - patches cause version explosion, compatibility nightmare
- Much better to eliminate security bugs at outset!

Vulnerabilities at CERT/CC

- The **CERT Coordination Center** `http://www.cert.org` at Carnegie Mellon University is a reporting centre for Internet security problems. They provide technical advice, recommended responses, identifying trends.

Vulnerabilities at CERT/CC

- The **CERT Coordination Center** `http://www.cert.org` at Carnegie Mellon University is a reporting centre for Internet security problems. They provide technical advice, recommended responses, identifying trends.
- Example statistics:

Year	2003	2004	2005	2006	2007	2008[Q1-3]
Vulnerabilities:	3784	3780	5990	8064	7236	6058

  The no. 1 category of vulnerabilities (over 50% up to 2004, at least) was the **buffer overflow**.

  More recently, with the rise of web applications written in higher-level languages, it has been taken over by **cross-site scripting** and **SQL injection**.

Categories of programming failure

1. **buffer overflow (inadequate input validation)**
2. **race conditions**
3. **access control mistakes**
4. **poor randomness**
5. **confidentiality leaks**

The following slides review each category.

- There are many check lists and emerging standard requirements for security checking, e.g. the **CWE/SANS Top 25** at `http://cwe.mitre.org/top25/`.
- Perhaps the single most important piece of advice for programming secure applications: **check your inputs!**

Outline

- Programming failures
- Buffer overflows
- Race conditions
- Permissions and Access Control
- Poor randomness
- Confidentiality leaks
- Building in security: design and guidelines

A few overflow vulnerabilities

splitvt, syslog, mount/umount, sendmail, lpr, bind, gethostbyname(), modstat, cron, login, sendmail again, the query CGI script, newgrp, AutoSofts RTS inventory control system, host, talkd, getopt(), sendmail yet again, FreeBSD s crt0.c, WebSite 1.1, rlogin, term, ffbconfig, libX11, passwd yppasswd nispasswd, imapd, ipop3d, SuperProbe, lpd, xterm, eject, lpd again, host, mount, the NLS library, xlock, libXt and further X11R6 libraries, talkd, fdformat, eject, elm, cxterm, ps, fbconfig, metamail, dtterm, df, an entire range of SGI programs, ps again, chkey, libX11, suidperl, libXt again, lquerylv, getopt() again, dtaction, at, libDtSvc, eeprom, lpr yet again, smbmount, xlock yet again, MH-6.83, NIS+, ordist, xlock again, ps again, bash, rdist, login/scheme, libX11 again, sendmail for Windows NT, wm, wwwcount, tgetent(), xdat, termcap, portmir, writesrv, rcp, opengroup, telnetd, rlogin, MSIE, eject, df, statd, at again, rlogin again, rsh, ping, traceroute, Cisco 7xx routers, xscreensaver, passwd, deliver, cidentd, Xserver, the Yapp conferencing server, . . .

A few overflow vulnerabilities – continued

multiple problems in the Windows95/NT NTFTP client, the Windows War and Serv-U FTP daemon, the Linux dynamic linker, filter (part of elm-2.4), the IMail POP3 server for NT, pset, rpc.nisd, Samba server, ufsrestore, DCE secd, pine, dslip, Real Player, SLMail, socks5, CSM, Proxy, imapd (again), Outlook Express, Netscape Mail, mutt, MSIE, Lotus Notes, MSIE again, libauth, login, iwsh, permissions, unfsd, Minicom, nslookup, zpop, dig, WebCam32, smbclient, compress, elvis, lha, bash, jidentd, Tooltalk, ttdbserver, dbadmin, zgv, mountd, pcnfs, Novell Groupwise, mscreen, xterm, Xaw library, Cisco IOS, mutt again, ospf_monitor, sdtcm_convert, Netscape (all versions), mpg123, Xprt, klogd, catdoc, junkbuster, SerialPOP, and rdist

- It’s frustrating that such a basic programming error can have such an enormous impact on software security, and doubly frustrating that it hasn’t been eliminated yet.

Source of buffer overflows

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

Source of buffer overflows

- 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.

Source of buffer overflows

- 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.

Source of buffer overflows

- 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.
- Overflows can corrupt other pieces of the program data and cause security bugs, or even execution of arbitrary code. . .

Smashing the stack for fun and profit

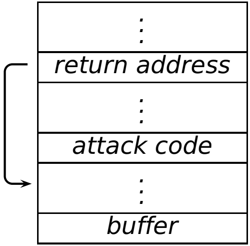
- Attack: exploit a program that uses a stack allocated buffer, by using a specially constructed longer-than-expected argument.

Smashing the stack for fun and profit

- Attack: exploit a program that uses a stack allocated buffer, by using a specially constructed longer-than-expected argument.
- The argument overwrites the return address, causing the CPU to execute part of it.

Smashing the stack for fun and profit

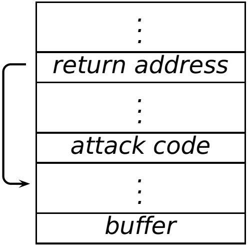
- Attack: exploit a program that uses a stack allocated buffer, by using a specially constructed longer-than-expected argument.
- The argument overwrites the return address, causing the CPU to execute part of it.



The malicious argument overwrites all of the space allocated for the buffer, all the way to the return address location. This 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.

Smashing the stack for fun and profit

- Attack: exploit a program that uses a stack allocated buffer, by using a specially constructed longer-than-expected argument.
- The argument overwrites the return address, causing the CPU to execute part of it.



The malicious argument overwrites all of the space allocated for the buffer, all the way to the return address location. This 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.

- Similar attacks work on the heap.

Fixing buffer overflows

- **good programming** to check bounds when necessary

Fixing buffer overflows

- **good programming** to check bounds when necessary
- **auditing** to find possible vulnerable code

Fixing buffer overflows

- **good programming** to check bounds when necessary
- **auditing** to find possible vulnerable code
- **special libraries** which contain bound-checking versions of standard functions

Fixing buffer overflows

- **good programming** to check bounds when necessary
- **auditing** to find possible vulnerable code
- **special libraries** which contain bound-checking versions of standard functions
- **StackGuard** compiler using “canaries”.

Fixing buffer overflows

- **good programming** to check bounds when necessary
- **auditing** to find possible vulnerable code
- **special libraries** which contain bound-checking versions of standard functions
- **StackGuard** compiler using “canaries”.
- **disabling stack or data execution**

Fixing buffer overflows

- **good programming** to check bounds when necessary
- **auditing** to find possible vulnerable code
- **special libraries** which contain bound-checking versions of standard functions
- **StackGuard** compiler using “canaries”.
- **disabling stack or data execution**
  - Execute Disable Bit (NX) now added to Intel and AMD CPUs

Fixing buffer overflows

- **good programming** to check bounds when necessary
- **auditing** to find possible vulnerable code
- **special libraries** which contain bound-checking versions of standard functions
- **StackGuard** compiler using “canaries”.
- **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

Outline

Race conditions

- Another technical attack: exploit *race conditions*.

Race conditions

- Another technical attack: exploit *race conditions*.
- Example: Unix `mkdir` used to work in two stages:

Race conditions

- Another technical attack: exploit *race conditions*.
- Example: Unix `mkdir` used to work in two stages:
  1. Make new directory
  2. Change ownership

Race conditions

- Another technical attack: exploit *race conditions*.
- 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.

Race conditions

- Another technical attack: exploit *race conditions*.
- 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.
- Race conditions can be hard to find, 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**.

Race conditions

- Another technical attack: exploit *race conditions*.
- 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.
- Race conditions can be hard to find, 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:
  - use locks in multi-threaded programming (`synchronized`)
  - reduce time-of-check, time-of-use (TOCTOU)

Outline

Permissions vulnerabilities

- Many exploits have taken advantage of failure to follow the ***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**:

Managing permissions

- Two extreme views:
  1. Most machines are single-user or single-application so user-level access controls don't matter. Separation of users lies in application-level code and network security.
  2. Trusted operating systems are vital, good security and strong access control mechanisms must be built-in to the lowest level.

Managing permissions

- Two extreme views:
  1. Most machines are single-user or single-application so user-level access controls don't matter. Separation of users lies in application-level code and network security.
  2. Trusted operating systems are vital, good security and strong access control mechanisms must be built-in to the lowest level.

  The first view was originally argued by vendors such as Microsoft and the second view was typical of the military.
  Nowadays, the second view is also being espoused by Microsoft and many other vendors (why?).

Outline

- Programming failures
- Buffer overflows
- Race conditions
- Permissions and Access Control
- Poor randomness
- Confidentiality leaks
- Building in security: design and guidelines

Poor randomness

- Numerous exploits have taken advantage of the predictability of supposedly “random” numbers.

Poor randomness

- Numerous exploits have taken advantage of the predictability of supposedly “random” numbers.
- Security applications require random numbers for various reasons, the most important of which is *key generation*.

Poor randomness

- Numerous exploits have taken advantage of the predictability of supposedly “random” numbers.
- Security applications require random numbers for various reasons, the most important of which is *key generation*.
- General strategy: **true random seed+cryptographically strong PRNG** 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.

Poor randomness

- Numerous exploits have taken advantage of the predictability of supposedly “random” numbers.
- Security applications require random numbers for various reasons, the most important of which is *key generation*.
- General strategy: **true random seed+cryptographically strong PRNG** 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.
- How do we get the true random seed? Without a dedicated random source, we must rely on non-deterministic external environmental data…

Environmental sources of randomness

- Good sources [RFC1750]: disk-head seek times, keystrokes, mouse movements, memory paging behaviour, network status, interrupt arrival times, random electrical noise (e.g. /dev/audio). Best use several, combined with a hash.

Environmental sources of randomness

- Good sources [RFC1750]: disk-head seek times, keystrokes, mouse movements, memory paging behaviour, network status, interrupt arrival times, random electrical noise (e.g. /dev/audio). Best use several, combined with a hash.
- Bad sources: system clock, Ethernet addresses or hardware serial numbers, network arrival packet timing or anything else that can be predicted or influenced by an adversary.

Environmental sources of randomness

- Good sources [RFC1750]: disk-head seek times, keystrokes, mouse movements, memory paging behaviour, network status, interrupt arrival times, random electrical noise (e.g. /dev/audio). Best use several, combined with a hash.
- Bad sources: system clock, Ethernet addresses or hardware serial numbers, network arrival packet timing or anything else that can be predicted or influenced by an adversary.
- Linux's random kernel device uses an "entropy pool" and estimates the number of "true" random bits in the pool. 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.

Outline

Storage confidentiality leaks

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

Storage confidentiality leaks

- Security applications store sensitive data in data-structures held in memory. Vulnerabilities:
  - Other processes may be able to read memory

Storage 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

Storage 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

Storage 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

Storage 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 sanitisation tricky

Storage 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 sanitisation tricky
  - Hard-disk data recovery can recover several generations of data

Storage 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 sanitisation tricky
  - Hard-disk data recovery can recover several generations of data
  - TEMPEST RF leakage from cables, monitors.

Storage 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 sanitisation tricky
  - Hard-disk data recovery can recover several generations of data
  - TEMPEST RF leakage from cables, monitors.
- Some defences:

Storage 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 sanitisation tricky
  - Hard-disk data recovery can recover several generations of data
  - TEMPEST RF leakage from cables, monitors.
- Some defences:
  - Touch memory regularly or OS calls to lock pages.

Storage 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 sanitisation tricky
  - Hard-disk data recovery can recover several generations of data
  - TEMPEST RF leakage from cables, monitors.
- Some defences:
  - Touch memory regularly or OS calls to lock pages.
  - Use custom swap file, zeroed after use.

Storage 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 sanitisation tricky
  - Hard-disk data recovery can recover several generations of data
  - TEMPEST 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

Storage 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 sanitisation tricky
  - Hard-disk data recovery can recover several generations of data
  - TEMPEST 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.

Outline

Security design principles

Saltzer and Schroeder (1975) gave 8 design principles as examples for OS protection (access control):

Security design principles

Saltzer and Schroeder (1975) gave 8 design principles as examples for OS protection (access control):

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.

Security design principles

Saltzer and Schroeder (1975) gave 8 design principles as examples for OS protection (access control):

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.

Security design principles

Saltzer and Schroeder (1975) gave 8 design principles as examples for OS protection (access control):

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.

Security design principles

Saltzer and Schroeder (1975) gave 8 design principles as examples for OS protection (access control):

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.

Security design principles — continued

5. **Separation of privilege** — 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.

Security design principles — continued

5. **Separation of privilege** — 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”.

Security design principles — continued

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

Security design principles — continued

5. **Separation of privilege** — 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”.
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.

Security design principles — continued

5. **Separation of privilege** — 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”.
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.

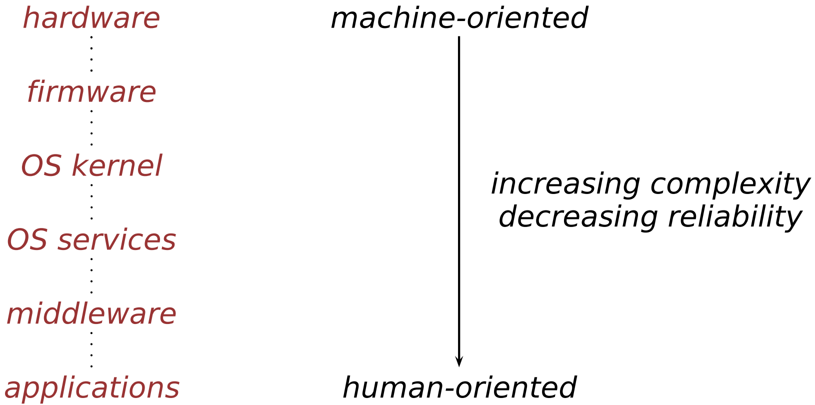
Security design principles — continued

5. **Separation of privilege** — 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”.
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.

Two further principles from physical security: **work factor** (comparison of cost of circumvention with the resources of an attacker) and **compromise recording** (make mechanisms tamper-evident).

Granularity of security provision

The hardware level has *fine grained* access controls. At higher levels, we implement increasingly user-oriented security policies. Reliability of each level depends on levels below, and increasingly complex implementations.



Programming principles

General principles for security programming emerge from case history of security vulnerabilities.

1. **Protect internal data and functions.** Use *language based access controls* (ADTs, visibility modifiers, modules).

Programming principles

General principles for security programming emerge from case history of security vulnerabilities.

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.

Programming principles

General principles for security programming emerge from case history of security vulnerabilities.

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).

Programming principles

General principles for security programming emerge from case history of security vulnerabilities.

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.

Some C coding guidelines (incomplete!)

- 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.

Some C coding guidelines (incomplete!)

- 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.
- Never use **`scanf`**; use `fgetc` (similarly, avoid `printf`, etc). In general, avoid routines which do not **check buffer boundaries**. Perform bounds checking on every array index when in any doubt.

Some C coding guidelines (incomplete!)

- 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.
- Never use **`scanf`**; use `fgetc` (similarly, avoid `printf`, etc). In general, avoid routines which do not **check buffer boundaries**. Perform bounds checking on every array index when in any doubt.
- Check **error return values**. Essential because C doesn't implement an exception mechanism.

Some C coding guidelines (incomplete!)

- 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.
- Never use **`scanf`**; use `fgetc` (similarly, avoid `printf`, etc). In general, avoid routines which do not **check buffer boundaries**. Perform bounds checking on every array index when in any doubt.
- Check **error return values**. Essential because C doesn't implement an exception mechanism.
- Don't keep **secret information in memory** of unprivileged programs; it may be possible to interrupt the program and cause it to dump core.

Some C coding guidelines (incomplete!)

- 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.
- Never use **`scanf`**; use `fgetc` (similarly, avoid `printf`, etc). In general, avoid routines which do not **check buffer boundaries**. Perform bounds checking on every array index when in any doubt.
- Check **error return values**. Essential because C doesn't implement an exception mechanism.
- Don't keep **secret information in memory** of unprivileged programs; it may be possible to interrupt the program and cause it to dump core.
- Consider **logging** UIDs, file accesses, etc..

Some C coding guidelines (incomplete!)

- 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.
- Never use **`scanf`**; use `fgetc` (similarly, avoid `printf`, etc). In general, avoid routines which do not **check buffer boundaries**. Perform bounds checking on every array index when in any doubt.
- Check **error return values**. Essential because C doesn't implement an exception mechanism.
- Don't keep **secret information in memory** of unprivileged programs; it may be possible to interrupt the program and cause it to dump core.
- Consider **logging** UIDs, file accesses, etc..
- **Strip binaries** (`strings` can reveal a lot!).

Some Unix coding guidelines (incomplete!)

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

Some Unix coding guidelines (incomplete!)

- 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.

Some Unix coding guidelines (incomplete!)

- 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.

Some Unix coding guidelines (incomplete!)

- 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).

Some Unix coding guidelines (incomplete!)

- 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 user-names.

References

- Mark G. Graff and Kenneth R. van Wyk. *Secure Coding: Principles & Practices*. O'Reilly, 2003.
- M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, second edition, 2003.
- John Viega and Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, 2001.
- John Viega and Matt Messier. *Secure Programming Cookbook for C and C++*. O'Reilly, 2003.
- David Wheeler. Secure Programming for Linux and Unix HOWTO. `http://www.dwheeler.com/secure-programs/`.

Recommended Reading

The short book Graff and van Wyk, or an equivalent, Chapters 1, 2, 5 of Wheeler.