T-110.5211 Cryptosystems

Requirements and design issues

6.11.2008

Kaufman et al: Chapter 26.23

Stallings: Chapter 7.4

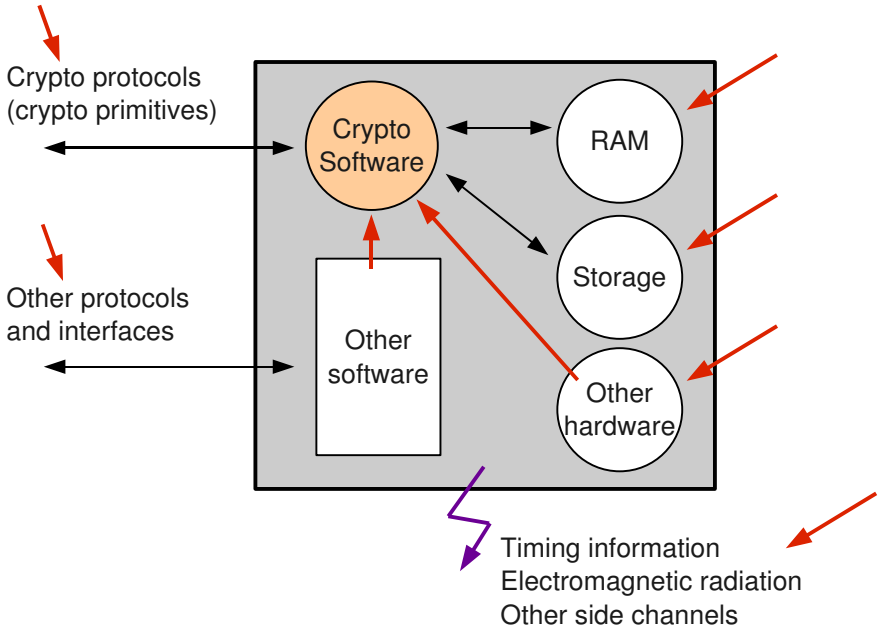
(There's very little in either book about these issues)

Outline

- Introduction to the second part of the course
- Requirements and design issues
  - Threats, security models, assumptions
  - Risk analysis
  - Choosing a cryptosystem
  - Assurance and evaluation
- Random number generation

Introduction to the second part of the course

Crypto implementation environment



Overview

- Security of crypto software is always critical
  - Crypto handles critical data (e.g. long term keys)
  - Leaks of just tens of bits may be catastrophic
- Security depends on protocols and implementation
  - “Mathematical”, software, and hardware security
  - Other components of hardware and software potential problems as well (e.g. crypto compromise through escalation)
- We'll look at cryptosystem software implementation
  - 1) Some generic issues in secure implementation
  - 2) Crypto specific implementation issues (and attacks)
- Much less organized than theory of cryptography
  - We'll proceed in software life cycle order (requirements, ...)

Security and functionality are very different

- The attacker always has the advantage
  - You need to cover all the holes, attacker needs to find **one** hole
- Security is invisible
  - You can't read it off a product label
  - You can't directly measure it, like functionality
- Security failures are often difficult to notice
  - Security failures are often silent
  - Example: encryption keys are not derived correctly
    - Proprietary crypto implementation initializes key to zero and only initializes the first byte of the key correctly => 256 possible keys
  - Example: cryptographic keys accidentally leaked to hard drive
    - Catastrophic for a hard disk encryption system
  - => Standards, interop testing with independent code, test vectors, etc.

What is a secure implementation?

- There are (at least) two broad concepts
  - 1) The program is free of flaws
  - 2) The program may have flaws, but they are not exploitable
- The second concept is what we use today
  - “Penetrate and patch” model of development
  - “Defense in depth”, multiple overlapping mechanisms
  - Damage control – fix flaws before they become public / exploitable
  - Continuous software updates
- The first concept is more or less theoretical
  - We don't need defense-in-depth if we operate in this model
  - Although theoretically possible, we lack the ability to write flawless software today, except in very small scale (and there are HW flaws, too)

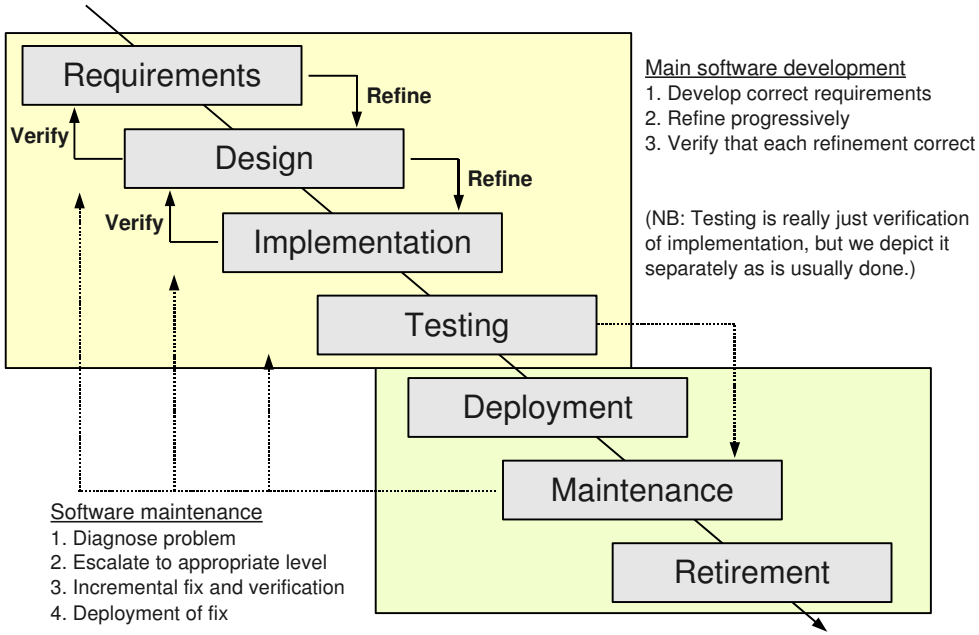
Some challenges

- We can't specify everything in practice
  - There are common expectations for secure software
  - Example: if result of function is **undefined** on some input, we still expect it to maintain security (what does that mean?)
  - Some of these are captured by “common design principles”, common expectations, defensive programming, etc
- **Secure** is relative to something, e.g. to a threat model
  - It does not mean there are no successful exploits
  - Example: physical tampering attack not in software scope
  - Example: RSA is **always breakable**
- **Secure** is often subjective
  - Example: estimating the expected monetary impact of an attack
  - What is the probability of an attack? What is the probability of success?
  - => Subjective estimates of probabilities

What keeps us from getting it right?

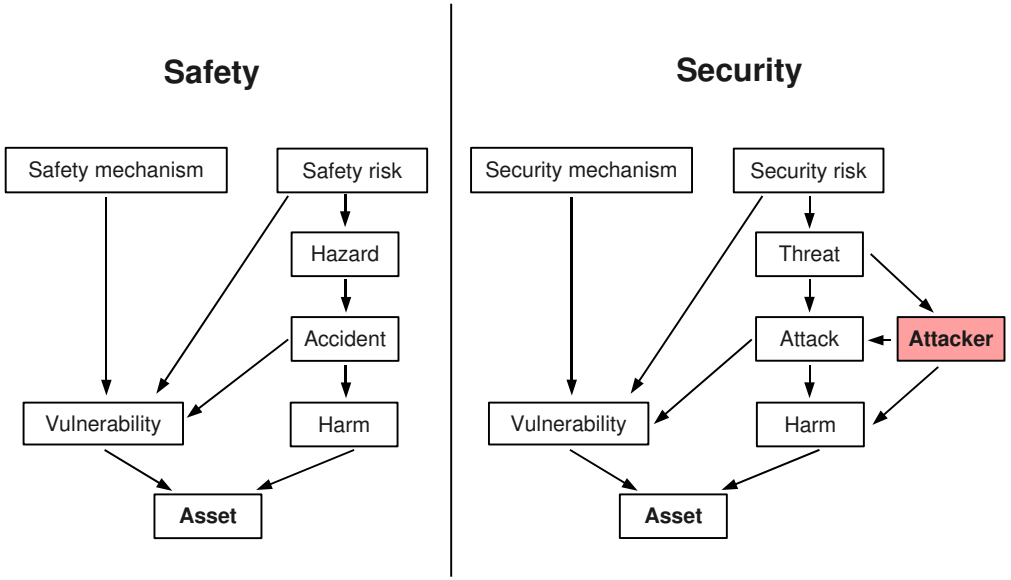
- Most of the world is focused on functionality – not security
  - These two are conceptually very different
- Programming languages and tools are not always suitable
  - "Unsafe" programming languages (e.g. C) widely used
  - "Safe" (type safe) programming languages exist, but they are not in widespread use at least in crypto programming
- Operating systems are not always suitable
  - High security crypto requires OS support
- Corporate pressure
  - "The release deadline is coming up, and the software works... "
- Security is not productive for a manager
  - Ideally security testing is a "waste of time" (= no flaws found)

Life cycle model



Requirements and design issues

Safety and security compared



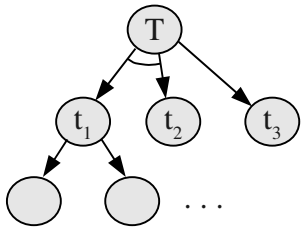
Assets can include people, financial assets, information, reputation, availability of resources, ...

Threat model

- Security is always relative to threats
  - Without a model of threats, difficult to define what security means
  - Threats are related to assets under protection (e.g. information)
- Defining the **threat model** is a good starting point
  - Define the threats for the system exhaustively (stepwise refinement)
  - Describe attackers and their capabilities
- The set of threats is infinite, so we try to capture relevant ones
  - Working on threat scenarios inspires better design
- Also rule out categories we're not interested in
  - E.g. physical tampering, electromagnetic attacks for networked software
- Threat model necessary even if you just apply standards
  - Your environment must match expectations of standard

Threat tree

- Threat tree
  - Root = all threats, children are main categories of threats
  - Nodes are further subdivided to more and more refined threats
  - Each level should be **complete =** cover all threats covered by parent
  - Often an **and**-**or** structure is used (notation varies)
- A part of threat model



Parent threat T is caused (only) by (t_1 **and** t_2) **or** t_3

Risk analysis

- Threats can be quantified using risk analysis
  - Cost of attack vs. expected damage to assets
  - Expected damage = Probability of success * Damage if success
  - Attacks are nearly always possible with enough money, so only a comparison to something (e.g. asset value) is sensible
- We can then make more informed decisions on security
  - Relevant and irrelevant threats
  - Threats which may become relevant if cost of attack decreases
- Notes about risks
  - 1. Risk is a function of environment (no attackers, no risk)
  - 2. Risks change with time (changes in env., new technology)
  - 3. Risks are often very remote, but still need to be considered. The more valuable your assets, the more remote risks you must consider.

Security objectives and assumptions

- Each threat is covered by
  - a) stating that it is out of scope (=> described in **assumptions**)
  - b) describing **security objective(s)** which address the threat
  - c) a combination of a) and b)
- Security objective (part of system)
  - A high level requirement, refined to security function(s) in design
- Assumption (external to system)
  - Example: Access to physical hosts restricted to trusted persons
- Example
  - **Threat –** unauthorized access to resources
  - **Security objective** – require user identification and auth. for access
  - **Assumption** – only network access possible, physical access is protected

Security functions

- More concrete than security objectives
  - E.g. cryptographic protocols and primitives
- Must address all security objectives satisfactorily
  - And ultimately, the threat model
- Example
  - **Threat** – passive eavesdroppers on communication line
  - **Security objective** – encryption for communication
  - **Security function** – anonymous TLS using ciphersuite X

Choosing a cryptosystem

- Dolev-Yao model is often good for networking
  - An attacker
    - can obtain any message passing through the network;
    - is a legitimate user of the network, and thus in particular can initiate a conversation with any other user;
    - will have the opportunity to be a receiver to any principal;
    - can send messages to any principal by impersonating any other principal.
  - In other words: the attacker carries your messages
- Network may often be considered only point of access
  - Assume physical security
  - Assume no “backdoors”
    - Login using SSH, use broken permissions to add trusted certificates
- => We're mostly interested in computational capabilities
  - Attacker computation capabilities, cryptosystem strength, key length, ...

Choosing a cryptosystem

- Layer in networking stack affects choice, examples
  - Bulk encryption (point-to-point) – IPsec / other packet-based
  - Application data (point-to-point) – TLS / other stream-based
  - Content (or relay protocol) – PGP or other object-security system
- Underlying layers are also your problem
  - Denial-of-Service is especially difficult
  - E.g. using TCP means you're vulnerable to all TCP threats
- Process and environment issues
  - Cryptosystems require a supporting environment (physical, processes, ...)
  - Key management is often problematic, e.g. huge PKI issue
  - User password management, dealing with lost passwords, etc.
  - If you don't take processes into accounts => problems
    - Example – people just accept **any** server certificates in HTTPS
    - => What is the true security increase provided by certificates?

Choosing a cryptosystem

- You should use an existing cryptosystem / protocol
  - Designing a cryptosystem is very difficult
  - Even if you get it right, you'll have to convince others
  - The gains of your own design are likely to be quite small
- You can use a profile of an existing cryptosystem
  - For instance: select a subset of crypto algorithms, fix key lengths, don't support optional functional features, etc.
  - You can even drop mandatory features if necessary, as long as you don't touch security critical components
  - Example: Our HTTPS client is a non-compliant subset, but interoperates

Choosing a cryptosystem

- Library support
  - If based on a standard, existing cryptosystems may have partial support (e.g. OpenSSL)
- Performance and code/data size
  - Sometimes cannot be overlooked, but less and less relevant
  - Code/data size can be minimized by e.g.
    - Decreasing number of supported algorithms
    - Selecting algorithms with small footprint
    - Specialized implementation for small size or performance
  - Asymmetric algorithms often use dynamic memory management, which may be a problem
    - Can be avoided by fixing big number sizes (key lengths)
  - Symmetric algorithms vary greatly in resource requirements
    - RC4 vs. 3DES

Assurance, trust, and evaluation

- Assurance point of view
  - We can't currently make software that remains secure over time
  - A system is **trustworthy**, if there is evidence that it meets given requirements; **trust** measures trustworthiness, relying on evidence
  - Assurance is more specific: **security assurance** is confidence that a system meets its security requirements, based on evidence provided by specific assurance techniques
- Categorization of assurance techniques
  - **Informal** – natural language for specs and justification of claims
  - **Semiformal** – natural language with a specific methodology
  - **Formal** – mathematics, formal languages, automated tools and techniques
- In practice, evaluation methods use semiformal approach

Assurance and trust

- Trust through evaluation
  - A system is considered trusted if it has been shown to meet requirements under an evaluation by certified experts
  - NB! Overloaded term, e.g. trusted system = system that we have to trust
- Note that trust and security are different concepts
  - A non-trusted system may be secure (just not evaluated)
  - A trusted system (evaluated) may be insecure
  - But we hope that evaluation correlates with overall security
- Evaluation provides assurance
  - Correspondence between life cycle stage deliverables
  - Requirements – Design – Implementation
- May also require minimum functional mechanisms
  - Increases trust that mechanisms are appropriate
  - E.g. a certain access control model, level of cryptographic strength

Some evaluation methodologies

- General purpose (access control, etc)
  - TCSEC (1983-1999) – DoD “Orange Book”
  - ITSEC (1991-2001) – European effort
  - Common Criteria (1998-present) – international
- FIPS 140-1 and 140-2 (1994-present)
  - Security Requirements for Cryptographic Modules
  - Security requirements cover 11 areas, including: basic design and documentation, key management, cryptographic algorithms, self testing
  - Although hardware oriented, may be used for software (on specific OS)

Formal methods

- Formal methods are one way to gain assurance
  - Correspondence between refinements can be proven to be correct
- Differences in many areas, e.g.
  - Underlying mathematical method
  - Degree of automation
  - Property verification or full verification
  - Domain of application – HW/SW, concurrent/sequential, etc
  - Pre- vs. post-development (design aid or verification)
- In practice can be applied only in limited scenarios
  - A certain module
  - A certain aspect of a module
- Informal and semiformal methods used in practice

Random number generation

Random number generator (RNG)

- Random number generator (RNG) is a critical primitive
  - Cryptographic protocols require good sources of randomness
    - Diffie-Hellman secret, random symmetric keys, RSA key gen. ...
    - Level of required strength in randomness varies
  - A bad random number generator may nullify security completely
    - Just as fatal as a broken protocol design, that's why we cover it
    - **Very** difficult to test, common implementation problem (e.g. PGP 2.6)
- Some fundamental problems
  - What is randomness and where to get it ?
  - Specific requirements for randomness in cryptographic use ?
  - Detecting a bad random number generator ?
- What is randomness ?
  - For our purposes, uncertain bits from attacker's perspective

Pseudorandom numbers

- Good statistical properties, e.g. for simulation and games
  - Example: Mersenne Twister
    - http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
- However, may be completely useless for cryptography
  - Pseudorandom number stream may reveal RNG seed value
  - => Attacker can easily reproduce stream and compute how the stream continues

“Real” randomness

- “Real” randomness – the best kind
  - Comes from an external source which we assume to be random (difficult to predict) and cannot be affected by attacker
  - Does not need to be unbiased, we can use e.g. hashing to make bit probability distribution unbiased
- Some examples
  - Background radiation
  - Lava lamp + webcam (SGI, http://www.lavarnd.org/)
    - Up to 200 kilobits of random data per second ! (reliability?)
- Less convincing sources
  - Hard drive seek and read/write characteristics
  - Interrupt timing at cycle level, keyboard and mouse input, ...
- Estimating uncertainty (entropy) of event is difficult

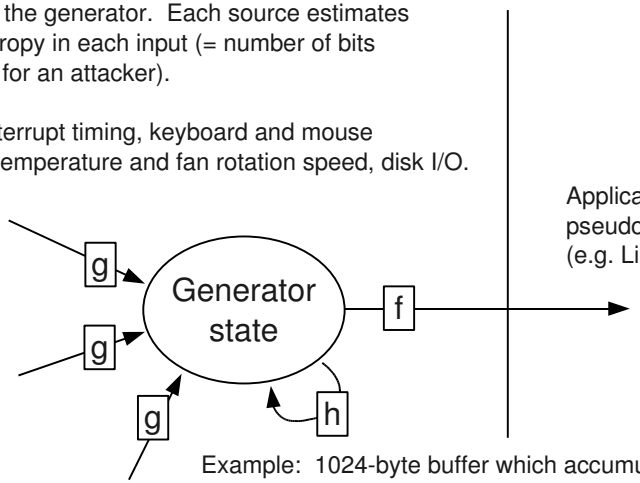
Cryptographic pseudorandom numbers

- Cryptographic pseudorandom number generator
  - Not just statistically random, but unpredictable without key (state)
  - => A small amount of real random bits (e.g. 128) can be extended to a much larger amount using a cryptographic PRNG
  - Given random bitstream, attacker should be unable to predict backwards or forwards
- Typical operation
  - Maintain state (“entropy pool”) with random data from various sources
  - Estimate the amount of entropy in the pool
    - E.g. every event comes with an estimate of # of random bits
  - Feed randomness until a certain threshold (# bits) is reached
    - Then start handing out data, but still keep feeding entropy
  - User data is extracted by hashing over the entropy pool (for example)

Cryptographic pseudorandom numbers

Multiple event sources with more or less uncertainty feed data into the generator. Each source estimates amount of entropy in each input (= number of bits unpredictable for an attacker).

Examples: interrupt timing, keyboard and mouse events, CPU temperature and fan rotation speed, disk I/O.



Applications extract pseudorandom bits (e.g. Linux /dev/urandom)

Example: 1024-byte buffer which accumulates randomness from events. If more than 1024 bytes available, current buffer is hashed through SHA1 to “compact” the buffer.
(Several variants are used: CRCs, XORing, ..)

Typical requirements

- After initial entropy gathering, unlimited bits available
  - We don't want applications to hang indefinitely (DoS problem)
- Brute forcing generator state computationally infeasible
  - Although doable with enough brute force power (2^{128} for instance)
- If generator state broken, damage should be minimized
  - Should not allow attacker to determine previous random numbers (“forward security”)
  - Generator should “escape” from attacker, i.e. re-establish a new state which the attacker cannot determine (“backward security”)

Typical requirements

- Should be resistant to malicious inputs
  - E.g. attacker generates network I/O events with little entropy
  - E.g. malicious kernel driver events with bogus entropy estimates
- Should not require good entropy estimates from sources
  - Driver writers will get it wrong

The “escape” mechanism

- A trivial scheme may not be able to “escape” once broken
  - If a trivial scheme is used, and all bits are fed into entropy pool directly, the requirement is not fulfilled
  - Attack: if e.g. 10 bits are fed into state, attacker extracts random bitstream, and brute forces the $\mathbf{2^{10}}$ possible new states, verifies brute force against observations => needs better design
- A slightly better mechanism
  - Gather entropy into a separate buffer w/o feeding to current state
  - Once enough gathered (e.g. 128 bits), feed all into current state at once
    - This forces attacker to try $\mathbf{2^{128}}$ brute force attack on the new state
  - Relies on knowing the amount of entropy in each event
- This can be improved still
  - Example (next slides) – Fortuna (Schneier & Ferguson)

Fortuna (Schneier & Ferguson)

- Background
  - Described in Schneier & Ferguson: *Practical Cryptography*
  - Improves on their previous design, Yarrow
  - Has not received extensive review, but it's an interesting design
- Basic design
  - **Generator**
    - Generates arbitrary amount of pseudorandom data based on a fixed size seed (state)
  - **Accumulator**
    - Gathers entropy from various sources and reseeds the generator once in a while
    - Does **not** need an entropy estimate
  - **Seed file**
    - Stores entropy across reboots to avoid slow PRNG startup

Fortuna – Generator

- Generator state
  - AES-256 key (**K**, 256 bits), Counter (**C**, 128 bits)
- Basically AES-256 counter mode with some twists
  - Pseudorandom stream is: $\mathbf{E_K(C{+}{+})} \mid \mathbf{E_K(C{+}{+})} \mid \mathbf{E_K(C{+}{+})}$ ...
- PRNG data is provided through individual PRNG requests
  - At most $\mathbf{2^{16}}$ blocks (1 megabyte) generated in one request
  - Generate requested amount of data (CTR), then generate new **K**
    - If attacker breaks current key, determining previous key (=> previous pseudorandom stream) is very difficult
    - The counter is not reset to avoid cycles (returning to same key/counter combination)
  - Counter mode is statistically biased (blocks don't repeat), which is why a limit on maximum data per key is imposed

Fortuna – Accumulator

- Reseed primitive – **Reseed(s)**, **s** is arbitrary data
  - **K ← SHA$_d$-256(K | s)** (*)
  - **C ← C+1** (uninitialized PRNG <=> **C=0**)
  - Used by accumulator to refresh current key
- Accumulator basics
  - Purpose – gather entropy from various sources and reseed PRNG
  - Works without knowing entropy per event – automatic adaptation
  - Uses the **Reseed(s)** primitive periodically or when enough entropy gathered
- Accumulator state
  - 32 entropy pools, **P0**, ..., **P31**
  - Reseed number, **r**

(*) SHA$_d$-256(x) = SHA-256(SHA-256(x)) (double hash)

Fortuna – Accumulator

- Feed randomness into pools
  - Randomness from each source is fed in a round-robin fashion to the 32 pools; a certain event goes to exactly one pool
- Reseed check
  - If **P$_0$** contains enough bytes (not **entropy**, bytes), start reseed op.
    - Also require that at least 100ms elapses before previous reseed
- Reseed operation
  - **r ← r + 1**
  - Find first bit set in **r** (starting from bit 0)
  - Include pools **P$_0$**, ..., **P$_k$** in reseed, where **k** is first bit set
  - Compute **s = SHA$_d$-256(P$_0$)** | ... | **SHA$_d$-256(P$_k$)**
  - Call **Reseed(s)**
  - Reset pools **P$_0$**, ..., **P$_k$** to empty strings

.... 1101$\underbrace{000}_{4}$

Fortuna – Accumulator

- Idea behind reseed operation
  - **P$_0$** included in every reseed, **P$_1$** every other, **P$_2$** every fourth, etc
- Guarantees escape
  - If entropy rate is large enough, **P$_0$** has enough entropy to "escape"
  - If entropy rate is lower, we eventually reseed with enough entropy
  - **P$_{31}$** is used every 10+ years, so it will gather a lot of entropy
- Entropy in each pool independent
  - Each event goes to a dedicated pool => pools are independent
    - Each event source is also independent, so malicious sources are OK
- Note clever form of pool hashing – **SHA$_d$-256(P$_i$)**
  - Data doesn't need to be stored, we maintain a **running SHA$_d$-256 hash**
  - Otherwise **P$_{31}$** could end up with gigabytes of data before reseed...

Fortuna – Seed file

- Seed file contains 64 bytes of randomness
  - Extracted from the generator
- On system startup
  - Atomic read-and-update function for seed file
    - Read seed file into **s**, check that it is exactly 64 bytes long
    - **Reseed(s)**
    - Overwrite seed file with 64 bytes from (reseeded) generator
  - Atomicity tries to ensure that on successive reboots the same seed is not reused if at all possible (tries to prevent e.g. power manipulation)
- Periodically, and on shutdown
  - Refresh seed file by writing 64 random bytes from generator
- Backups are a problem also for Fortuna seed files

Linux 2.6.8 /dev/[u]random

- Linux exposes two userspace cryptographic PRNG devices
  - **/dev/random** – "true" randomness, entropy not reused
  - **/dev/urandom** – pseudo randomness, entropy reused if necessary
- Basic design (partially based on PGP)
  - Maintains an entropy pool + entropy estimate (max pool size)
  - Input – stir pool using a CRC-32, rotate+twist bits (ultimately XOR)
    - Input is batched and inputed separately (avoids interrupt latency)
  - Entropy estimates are more or less heuristic
    - Timers: $\mathbf{log_2\ [min(|delta|, |delta^2|, |delta^3|) / 2]}$, cap to **12**
    - Event data (keyboard scancode, mouse position etc) => also added
  - Output – use SHA1 to extract entropy, update entropy estimate
    - Reads on **/dev/random** block until (estimated) entropy available
- Not necessarily very secure (*)

(*) Gutterman, Pinkas, Reinman: *Analysis of the Linux Random Number Generator*, March 2006 [kernel 2.6.10]
http://eprint.iacr.org/2006/086.pdf

Resources

- RFC 1750: Randomness Recommendations for Security
  - A good introduction to the various issues, but does not contain actual algorithms
- Schneier & Ferguson: *Practical Cryptography*
  - Contains a description of the Fortuna algorithm
- Existing cryptographic PRNGs
  - PGP
  - Linux /dev/[u]random
  - ...

Summary

- Introduction to the second part of the course
- Requirements and design issues
  - Threats, security models, assumptions
  - Risk analysis
  - Choosing a cryptosystem
  - Assurance and evaluation
- Random number generation