Short history of SSH
- Started out as freeware by Tatu Ylönén in 1995
- Original version commercialized
- Fully open-source OpenSSH from OpenBSD
- Protocol redesigned and standardized for "SSH 2"

SSH host keys
- Every SSH server has a public/private keypair
- Ideally, never changes once SSH is installed
- Early generation a classic entropy problem
  - Especially embedded systems, VMs

Authentication methods
- Password, encrypted over channel
- .shosts: like .rhosts, but using client host key
- User-specific keypair
  - Public half on server, private on client
- Plugins for Kerberos, PAM modules, etc.

Old crypto vulnerabilities
- 1.x had only CRC for integrity
  - Worst case: when used with RC4
- Injection attacks still possible with CBC
  - CRC compensation attack
- For least-insecure 1x-compatibility, attack detector
- Alas, detector had integer overflow worse than original attack

Newer crypto vulnerabilities
- IV chaining: IV based on last message ciphertext
  - Allows chosen plaintext attacks
  - Better proposal: separate, random IVs
- Some tricky attacks still left
  - Send byte-by-byte, watch for errors
  - Of arguable exploitability due to abort
- Now migrating to CTR mode
SSH over SSH
- SSH to machine 1, from there to machine 2
  - Common in these days of NATs
- Better: have machine 1 forward an encrypted connection
  1. No need to trust 1 for secrecy
  2. Timing attacks against password typing

SSH (non-)PKI
- When you connect to a host freshly, a mild note
- When the host key has changed, a large warning

Outline
- SSH
- SSL/TLS
- More causes of crypto failure
- Software engineering for security

SSL/TLS
- Developed at Netscape in early days of the public web
  - Usable with other protocols too, e.g. IMAP
- SSL 1.0 pre-public, 2.0 lasted only one year, 3.0 much better
- Renamed to TLS with RFC process
  - TLS 1.0 improves SSL 3.0
  - TLS 1.1 and 1.2 in 2006 and 2008, only gradual adoption

IV chaining vulnerability
- TLS 1.0 uses previous ciphertext for CBC IV
- But, easier to attack in TLS:
  - More opportunities to control plaintext
  - Can automatically repeat connection
- "BEAST" automated attack in 2011: TLS 1.1 wakeup call

Compression oracle vuln.
- Compr($S \ || \ A$), where $S$ should be secret and $A$ is attacker-controlled
- Attacker observes ciphertext length
  - If $A$ is similar to $S$, combination compresses better
- Compression exists separately in HTTP and TLS

But wait, there’s more!
- Too many vulnerabilities to mention them all in lecture
- Kaloper-Meršinjak et al. have longer list
  - "Lessons learned" are variable, though
- Meta-message: don’t try this at home

HTTPS hierarchical PKI
- Browser has order of 100 root certs
  - Not same set in every browser
  - Standards for selection not always clear
- Many of these in turn have sub-CAs
- Also, “wildcard” certs for individual domains
Hierarchical trust?
- No. Any CA can sign a cert for any domain
- A couple of CA compromises recently
- Most major governments, and many companies you’ve never heard of, could probably make a google.com cert
- Still working on: make browser more picky, compare notes

CA vs. leaf checking bug
- Certs have a bit that says if they’re a CA
- All but last entry in chain should have it set
- Browser authors repeatedly fail to check this bit
- Allows any cert to sign any other cert

MD5 certificate collisions
- MD5 collisions allow forging CA certs
- Create innocuous cert and CA cert with same hash
  - Requires some guessing what CA will do, like sequential serial numbers
  - Also 200 PS3s
- Oh, should we stop using that hash function?

CA validation standards
- CA’s job to check if the buyer really is foo.com
- Race to the bottom problem:
  - CA has minimal liability for bad certs
  - Many people want cheap certs
  - Cost of validation cuts out of profit
- “Extended validation” (green bar) certs attempt to fix

HTTPS and usability
- Many HTTPS security challenges tied with user decisions
- Is this really my bank?
- Seems to be a quite tricky problem
  - Security warnings often ignored, etc.
  - We’ll return to this as a major example later

Outline
- SSH
- SSL/TLS
- More causes of crypto failure
- Software engineering for security

Random numbers and entropy
- Cryptographic RNGs use cipher-like techniques to provide indistinguishability
- But rely on truly random seeding to stop brute force
  - Extreme case: no entropy → always same “randomness”
- Modern best practice: seed pool with 256 bits of entropy
  - Suitable for security levels up to $2^{256}$

Netscape RNG failure
- Early versions of Netscape SSL (1994-1995) seeded with:
  - Time of day
  - Process ID
  - Parent process ID
- Best case entropy only 64 bits
  - (Not out of step with using 40-bit encryption)
- But worse because many bits guessable
Debian/OpenSSL RNG failure (1)
- OpenSSL has pretty good scheme using /dev/urandom
- Also mixed in some uninitialized variable values
  - "Extra variation can't hurt"
- From modern perspective, this was the original sin
  - Remember undefined behavior discussion?
- But had no immediate ill effects

Debian/OpenSSL RNG failure (2)
- Debian maintainer commented out some lines to fix a Valgrind warning
  - "Potential use of uninitialized value"
- Accidentally disabled most entropy (all but 16 bits)
- Brief mailing list discussion didn't lead to understanding
- Broken library used for ~2 years before discovery

Detected RSA/DSA collisions
- 2012: around 1% of the SSL keys on the public net are breakable
  - Some sites share complete keypairs
  - RSA keys with one prime in common (detected by large-scale GCD)
- One likely culprit: insufficient entropy in key generation
  - Embedded devices, Linux /dev/urandom vs. /dev/random
- DSA signature algorithm also very vulnerable

Newer factoring problem (CCS'17)
- An Infineon RSA library used primes of the form
  \[ p = k \cdot M + (65537 \cdot a \mod M) \]
- Smaller problems: fingerprintable, less entropy
- Major problem: can factor with a variant of Coppersmith’s algorithm
  - E.g., 3 CPU months for a 1024-bit key

Side-channel attacks
- Timing analysis:
  - Number of 1 bits in modular exponentiation
  - Unpadding, MAC checking, error handling
  - Probe cache state of AES table entries
- Power analysis
  - Especially useful against smartcards
- Fault injection
- Data non-erasure
  - Hard disks, "cold boot" on RAM

WEP “privacy”
- First WiFi encryption standard: Wired Equivalent Privacy (WEP)
- F&S: designed by a committee that contained no cryptographers
- Problem 1: note "privacy": what about integrity?
  - Nope: stream cipher + CRC = easy bit flipping

WEP shared key
- Single key known by all parties on network
- Easy to compromise
- Hard to change
- Also often disabled by default
- Example: a previous employer

WEP key size and IV size
- Original sizes: 40-bit shared key (export restrictions) plus 24-bit IV = 64-bit RC4 key
  - Both too small
- 128-bit upgrade kept 24-bit IV
  - Vague about how to choose IVs
  - Least bad: sequential, collision takes hours
  - Worse: random or everyone starts at zero
WEP RC4 related key attacks

- Only true crypto weakness
- RC4 "key schedule" vulnerable when:
  - RC4 keys very similar (e.g., same key, similar IV)
  - First stream bytes used
- Not such a problem for other RC4 users like SSL
  - Key from a hash, skip first output bytes

Newer problem with WPA (CCS’17)

- Session key set up in a 4-message handshake
- Key reinstallation attack: replay #3
  - Causes most implementations to reset nonce and replay counter
  - In turn allowing many other attacks
  - One especially bad case: reset key to 0
- Protocol state machine behavior poorly described in spec
  - Outside the scope of previous security proofs

Trustworthiness of primitives

- Classic worry: DES S-boxes
- Obviously in trouble if cipher chosen by your adversary
- In a public spec, most worrying are unexplained elements
- Best practice: choose constants from well-known math, like digits of \( \pi \)

Dual_EC_DRBG (1)

- Pseudorandom generator in NIST standard, based on elliptic curve
- Looks like provable (slow enough!) but strangely no proof
- Specification includes long unexplained constants
- Academic researchers find:
  - Some EC parts look good
  - But outputs are statistically distinguishable

Dual_EC_DRBG (2)

- Found 2007: special choice of constants allows prediction attacks
  - Big red flag for paranoid academics
- Significant adoption in products sold to US govt. FIPS-140 standards
  - Semi-plausible rationale from RSA (EMC)
- NSA scenario basically confirmed by Snowden leaks
  - NIST and RSA immediately recommend withdrawal

Outline

SSH
SSL/TLS
More causes of crypto failure
Software engineering for security

Defensive programming

- Analogy to defensive driving: drive so that there won't be a crash even if other drivers are negligent
- Don't just avoid bugs, reduce risks
- Aim for security even if other code and programmers are imperfect

Modularity

- Divide software into pieces with well-defined functionality
- Isolate security-critical code
  - Minimize TCB, facilitate privilege separation
  - Improve auditability
Minimize interfaces
- Hallmark of good modularity: clean interface
- Particularly difficult:
  - Safely implementing an interface for malicious users
  - Safely using an interface with a malicious implementation

Appropriate paranoia
- Many security problems come down to missing checks
- But, it isn't possible to check everything continuously
- How do you know when to check what?

Invariant
- A fact about the state of a program that should always be maintained
- Assumed in one place to guarantee in another
- Compare: proof by induction

Pre- and postconditions
- Invariants before and after execution of a function
- Precondition: should be true before call
- Postcondition: should be true after return

Dividing responsibility
- Program must ensure nothing unsafe happens
- Pre- and postconditions help divide that responsibility without gaps

When to check
- At least once before any unsafe operation
- If the check is fast
- If you know what to do when the check fails
- If you don't trust
  - your caller to obey a precondition
  - your callee to satisfy a postcondition
  - yourself to maintain an invariant

Sometimes you can’t check
- Check that \( p \) points to a null-terminated string
- Check that \( fp \) is a valid function pointer
- Check that \( x \) was not chosen by an attacker

Error handling
- Every error must be handled
  - i.e. program must take an appropriate response action
- Errors can indicate bugs, precondition violations, or situations in the environment
### Error codes
- Commonly, return value indicates error if any
- Bad: may overlap with regular result
- Bad: goes away if ignored

### Exceptions
- Separate from data, triggers jump to handler
- Good: avoid need for manual copying, not dropped
- May support: automatic cleanup (finally)
- Bad: non-local control flow can be surprising