Using Cryptography
We’ve covered a lot of cryptography principles—but how do we actually use it?

Beyond the basics—don’t invent your own algorithms or protocols—what are the issues in practice?

Lots of them...
Random numbers are vital for cryptography
They’re used for keys, nonces, primality testing, and more
Where do they come from?
What is a Random Number?

- Must be *unpredictable*
- Must be drawn from a large-enough space
- Ordinary statistical-grade random numbers are not sufficient
- *Distribution* not an indication of randomness: loaded dice are still random!
Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.

—John von Neumann, 1951
Sources of Random Numbers

- Dedicated hardware random number sources
- Random numbers lying around the system
- Software pseudo-random generator
- Combinations
Hardware Random Number Generators

- Radioactive decay
- Thermal noise
- Oscillator pairs
- Other chaotic processes
Radioactive Decay

- Timing of radioactive decay unpredictable even in theory—it’s a quantum process
- Problem: low bit rate from rational quantities of radioactive material
- Problem: not many computers have Geiger counters or radioactive isotopes attached...
- See [http://www.fourmilab.ch/hotbits/hardware3.html](http://www.fourmilab.ch/hotbits/hardware3.html) for a description of how to do it...
Thermal Noise

- Any electronic device has a certain amount of random noise (thermal noise in the components)
- Example: Take a sound card with no microphone and turn up the gain to maximum
- Or use a digital camera with the lens cap on
- Problem: modest bit rate
Oscillator Pairs

- Have a free-running fast R-C oscillator (don’t use a crystal; you don’t want it accurate or stable!)
- Have a second, much slower oscillator
- At each zero-crossing of the slow oscillator, sample the value of the fast oscillator
- Caution: watch for correlations or couplings between the two
Other Chaotic Processes

- Mouse movements
- Keystroke timing (low-order bits)
- Network packet timing (low-order bits)
- Disk seek timing: air turbulence affects disk internals (but what about solid state disks?)
- At boot time, there’s not much of this available
- Also: what if the enemy can observe the process?
- Cameras and Lava Lites®! (http://www.lavarnd.org/)
Problems

- Need deep understanding of underlying physical process
- Stuck bits
- Variable bit rate
- How do we measure their randomness?
- Assurance—how do we know it’s working properly?
Again, ordinary generators, such as C’s `random()` function or Java’s `Random` class are insufficient.

Can use cryptographic primitives—encryption algorithms or hash functions—instead.

But—where does the seed come from?
Generating Strong Pseudo-Random Numbers?

```c
unsigned int nextrand()
{
    static unsigned int state;
    static int first = 1;

    if (first) {first = 0; state = truerand();}
    state = f(state);
    return sha256(state);
}
```

- State is initialized from a true-random source
- Can’t invert sha256() to find state from return value
- But there is a serious problem here. What is it?
sha256() isn’t invertible, but we can do a brute force attack
state is too short; we can try all possible values in $2^{32}$ iterations
Estimated resources on a 3.4 Ghz Pentium: 3.6 hours CPU time; 150 GB to store all of the values
The attack parallelizes nicely
Need enough state—and hence enough true-random bits—that brute force is infeasible.
An application can keep a file with a few hundred bytes of random numbers

Generate some true-random bytes, mix with the file, and extract what you need

Write the file back to disk—read-protected, of course—for next time

What about stored VMs? Will they get the same seed each time?

Also: “mixing” isn’t as easy as it sounds
Many operating systems can provide cryptographic-grade random numbers

/dev/random: True random numbers, from hardware sources (but don’t use it!)
/dev/urandom: Software random number generator, seeded from hardware

Windows: CryptGenRandom()—similar to /dev/urandom

And there are APIs—in Python 3, use the secrets class instead of random
A Well-Known Failure

- As noted, not much randomness is available at boot time
- But—that’s often when key pairs are generated
- An RSA public key is the product of two “random” primes
- Might one be predictable?
- Heninger, Durumeric, Wustrow, and Halderman showed that many ssh keys have at least one predictable prime factor, for just this reason
- The same thing happened with several countries’ national ID cards
NIST decided to standardize a software PRNG

This is a good thing

NIST picked several designs—and the NSA persuaded NIST to include another based on elliptic curve cryptography

It seemed odd—DUAL_EC is quite slow, since it’s based on public key technology—but the NSA insisted that they needed it. They did need it, but not for the usual reason...

At least one company, RSA, made it the default in their product, allegedly after being paid off

Juniper used it in their routers—unclear why
The Problem with DUAL_EC_DRBG

- The algorithm includes a “random” constant
- If it’s not random—if it’s the public key in an elliptic curve cryptosystem—anyone who can see enough of the output from the PRNG and knows the corresponding private key can predict all future output from the algorithm
- Many protocols do in fact transmit some random bits in the clear
- There have been public demonstrations that it’s exploitable under certain circumstances
- Does the NSA know the corresponding private key? They’ve never said...
- *Someone*—supposedly not Juniper—changed the magic constant in Juniper’s version. Do they know the new private key?
- NIST has removed DUAL_EC_DRBG from their standard, RSA has removed it from their code...
Hardware Versus Software
Random Number Generators

- Hardware values can be true-random
- Output rate is rather slow
- Subject to environmental malfunctions, such as 60 Hz noise
- Software, if properly designed and written, is fast and reliable
- Combination of software generator with hardware seed is usually best
To paraphrase Knuth, random numbers should not be generated by a random process.

In many systems, hardware and software, random number generation is a very weak link.

Use standard facilities when available; if not, pay attention to RFC 4086.
Data in motion: protect a communications session
Data at rest: protect a file or device
The properties are very different
Both parties are present for the cryptographic protocol

Certain items can be negotiated, such as which algorithms are supported

Confidentiality must be future-proof; authenticity generally need not be—authenticity only matters during the life of the session
Using Cryptography

Protecting Files: Data at Rest

- Encryption and decryption are asynchronous; you don’t know when the decryption will take place.
- In the future, no idea which algorithms will be supported (old, insecure algorithms are often deleted from programs).
- Authenticity may be an issue, if you have to verify in the future that the file is genuine.
Protecting Files — Issues

- Suppose we want to use crypto to protect files. Now what?
- What to encrypt?
- Where should keys be stored?
- What is the tradeoff between availability and confidentiality?
Why Encrypt Files?

- Theft of files
- Theft of backup media
- Theft of computer
Is there a flaw in the operating system’s protection mechanisms? Why can’t the OS keep bad guys from the file?

Do you trust your sysadmin?

Are you using a cloud VM? What about the cloud sysadmin?

Laptops have feet — a remarkably high percentage are stolen
September 17, 2000
IRVINE – Qualcomm founder Irwin Jacobs’ laptop computer disappeared during a conference yesterday in an apparent theft that could put some of the company’s most sensitive secrets at risk.

Jacobs said his laptop contained "everything," secret corporate information, including e-mail dating back years, financial statements and even personal mementos.

Though Jacobs’ IBM ThinkPad PC is valued at about $3,700, the value of the information it contained is incalculable to Qualcomm and to Jacobs.
Caveats

- File encryption can help
- But there may be a serious convenience issue
- It may result in a *loss* of availability, if you lose the key
Encryption Options

- Manually encrypt/decrypt files
- Encrypt an entire disk or partition
Manual Encryption

- Very inconvenient to use
- Users are constantly supplying keys
- Most utilities won’t have direct interfaces to the decryption function; you have to manually decrypt files before use
- Users *will* forget to re-encrypt files
- Important design principle: *make it easy for users to do the right thing*
Encrypt an entire disk or disk partition
- Protects everything, even the free space
- Very important, given that “delete” operations do not delete the data
- Useful for protecting swap area
- Built into Windows (BitLocker) and MacOS (FileVault)
- Pretty much ubiquitous on modern phones
Decrypting the Disk

- Encrypt it? Where does the decryption key come from?
- One answer: supplied at reboot time
- In a USB drive plugged into a server?
- Tradeoff: availability versus confidentiality and integrity
- Use secure crypto hardware to decrypt database?
- Who has what sort of access, and what are their powers?
How Does a User Store a Key?

- Store key on disk, encrypted
- Generally decrypted with passphrase
- Passphrases are weak, but they’re a second layer, on top of OS file access controls
- Special-purpose hardware
- Or—convert a passphrase directly to a key
Secure Cryptographic Hardware

- HSM—Hardware Security Module
- Can be used for users or servers
- More than just key storage; perform actual cryptographic operations
- Enemy has *no* access to secret or private keys
- Friends have no access, either
- Modular exponentiation can be done much faster with dedicated hardware
Many PCs have TPM—Trusted Platform Module—chips
Newer Macs have Apple’s T2 chip
iPhones use a “secure enclave” in the CPU
iOS Encryption

- At first boot, the phone generates an internal AES-256 key
- This key remains within the secure enclave and can’t be exported
- The user’s PIN is converted to an AES-256 key using PBKDF2 (stay tuned); this PIN-derived key is mixed with the internal key inside the secure enclave to produce a master key
- By default, there are limited retries on PIN entry
- All storage is encrypted, sometimes with the internal key, sometimes with the master key, and sometimes with a new key derived from the master key
- All that happens inside the secure enclave—and without the PIN and master key, you can’t decrypt anything...
iOS Encryption

- At first boot, the phone generates an internal AES-256 key
- This key remains within the secure enclave and can’t be exported
- The user’s PIN is converted to an AES-256 key using PBKDF2 (stay tuned); this PIN-derived key is mixed with the internal key inside the secure enclave to produce a master key
- By default, there are limited retries on PIN entry
- All storage is encrypted, sometimes with the internal key, sometimes with the master key, and sometimes with a new key derived from the master key
- All that happens inside the secure enclave—and without the PIN and master key, you can’t decrypt anything...
- ...supposedly—in reality, there have been bugs
Hardware Issues

- Hardware must resist physical attack
- Environmental sensors: detect attack and erase keys
  - Example: surround with wire mesh of known resistance; break or short circuit is detected
  - Example: temperature sensor, to detect attempt to freeze battery
Tamper-resistant, not tamper-proof

Again: who is your enemy, and what are your enemy’s powers?

How does Alice talk to it securely? How do you ensure that an enemy doesn’t talk to it instead?

What is Alice’s intent? How does the crypto box know?

What if there are bugs in the cryptographic processor software? (IBM’s 4758 has a 486 inside. That can run complex programs…)

Research shows that most HSMs are, in fact, insecure
Different machine-level operations can take different amounts of time.
- Fetching data from the cache is much faster than fetching it from RAM.
- This can be used by attackers to learn a key!
- Example: suppose the attacker is on the same physical machine as you in a cloud datacenter.
- Sometimes, such attacks can even be done remotely.
Passphrases as Keys

- Passphrases are lousy keys—people pick bad ones, reuse them, they don’t have enough entropy, and more
- Sometimes, though, they’re all we have
- Goals: make the key look pseudo-random and impede guessing attempts
- Techniques: hash functions, iteration (at least 10,000 times), salt—the salt is a 128-bit or longer random number
- More on these techniques next class
If we need $l$ bits of keying material and our hash function emits $h$-bit values, we need $n = \lceil l/h \rceil$ invocations of $F$. If $P$ is the password, $s$ is the salt, and $c$ is the iteration count:

$$k = F(P, s, c, 1) \parallel F(P, s, c, 2) \parallel \ldots \parallel F(P, s, c, n)$$

$$F(P, s, c, i) = U_1 \oplus U_2 \oplus \ldots \oplus U_c$$

where:

$$U_1 = H(P, s \parallel \text{int32}(i))$$
$$U_2 = H(P, U_1)$$
$$\ldots$$
$$U_c = H(P, U_{c-1})$$

That is: run a hash function over $P$, $s$ and the block counter $i$ iterated $c$ times, exclusive-ORing the outputs together, for each portion of $k$. 

PBKDF2: Password-Based Key Derivation Function 2
Why This?

- The salt means that two uses of the same password will produce different keys.
- But—the salt must be available to the decryptor. (For file or disk encryption, that means storing the salt with the encrypted file.)
- Using many iterations slow down guessing.
Using the Generated Key

- Pick a random key to encrypt the data (DEK—Data-Encrypting Key)
- In fact, generate multiple DEKs, one for each section of the disk
- Use the user-supplied key to encrypt the DEK

♫ This makes changing the password fast
- (Effectively) erasing the disk is also very quick—just overwrite the DEK
Key Expansion

- Suppose you need multiple keys derived from one original key, either from a password or from a Diffie-Hellman exchange.

- We use *key expansion*.

- For each key you need, pick a label $L$, perhaps “C” for a confidentiality key, “I” for an integrity key, etc.

- Then $K_L = H(K, L)$, where $K$ is the original key and $H$ is a cryptographic hash function.
Protecting In-Memory Keys

- If a key is in RAM, it can be stolen from there
- If it’s swapped out to disk, it can persist on the disk—use the mlock() system call to lock it into RAM
- When a key (or the password it is derived from) is no longer needed, zero the memory—but that’s trickier than it looks

```c
char k[32];
get_key(k);
decrypt_file(k, filename);
memset(k, (char) 0, sizeof k);
return;
```
Protecting In-Memory Keys

- If a key is in RAM, it can be stolen from there.
- If it’s swapped out to disk, it can persist on the disk—use the `mlock()` system call to lock it into RAM.
- When a key (or the password it is derived from) is no longer needed, zero the memory—but that’s trickier than it looks.

```c
char k[32];
get_key(k);
decrypt_file(k, filename);
memset(k, (char) 0, sizeof k);
return;
```

- What’s wrong?

A good optimizing compiler will realize that `k` is not used after zeroing, and will optimize away the call.
Protecting In-Memory Keys

- If a key is in RAM, it can be stolen from there
- If it’s swapped out to disk, it can persist on the disk—use the mlock() system call to lock it into RAM
- When a key (or the password it is derived from) is no longer needed, zero the memory—but that’s trickier than it looks

```c
char k[32];
get_key(k);
decrypt_file(k, filename);
memset(k, (char) 0, sizeof k);
return;
``` 

What’s wrong?

- A good optimizing compiler will realize that `k` is not used after zeroing, and will optimize away the call...
(American kestrel, Morningside Park, September 22, 2020)