Pseudorandom Number Generation

Thanks once again to A. Joseph, D. Tygar, U. Vazirani, and D. Wagner at the University of California, Berkeley

What Can Go Wrong?

- An example:

```
unsigned char key[16];

srand(time(NULL));
for (i=0; i<16; i++) {
  key[i] = rand() & 0xFF;
}
```

- This generates a 16 byte (128 bit) key

A Refresher (?)

- The function prototypes

```
int rand(void);
void srand(unsigned int seed);
time_t time(time_t *t);
```

- Each call to **`rand()`** returns pseudorandom number with values in range 0 to **`RAND_MAX`**, calculated as deterministic function of the seed.
- **`srand(s)`** sets the seed to s
- **`time(NULL)`** returns current time, seconds since Jan 1, 1970.

Possible Implementations

```c
static unsigned int next = 0;
void srand(unsigned int seed) {
   next = seed;
}

/* RAND_MAX assumed to be 32767 */
int rand(void) {
   next = next * 1103515245 + 12345;
   return next % 32768;
}
```

A Problem: Key is easily guessed

- Seed is highly predictable
  - If Alice generates new session key at start of each session, then anyone who eavesdrops on session can determine (within small range) the time of day on Alice's machine.
  - Even if only narrow time to within one year, there are only 3600 x 24 x 365 = 31,536,000 ≈ 2^{25} keys. Modern machines can try all within minutes
- Algorithm used by **`rand()`** is publicly known
- If you can guess seed and know the algorithm, you have the key

Another Problem: Output is non-random

- In fact, it's highly non-random
  - E.g., low bit of every successive output of rand() alternates (0,1,0,1,0,1,…). (Why?)
    - Thus key space has been reduced from 2^{128} to 2^{113}. (Why?)

Another Problem: Output is non-random

- Each output of rand() depends only on previous output of rand()
  - Let N_0, N_1, N_2,… be sequence of `next` values during successive calls to **`rand().`** On 32-bit machine we have $N_{i+1} = 1103515245 \times N_i + 12345$
  - Let X_i be output of ith call to **`rand().`** Then $X_i = N_i \pmod{2^{15}}$.
  - Thus $x_{i+1} = (1103515245 \times X_i + 12345) \bmod 2^{15}$

Assume N_0 is the seed. Then

$$
\begin{aligned}
X_1 &= N_1 \bmod 2^{15} \\
N_2 &= N_1 * 1103515245 + 12345 \\
X_2 &= N_2 \bmod 2^{15} \\
&= (N_1 * 1103515245 + 12345) \bmod 2^{15} \\
&= (X_1 * 1103515245 + 12345) \bmod 2^{15} \\
N_3 &= (N_2 * 1103515245 + 12345) \bmod 2^{15} \\
X_3 &= N_3 \bmod 2^{15} \\
&= (N_2 * 1103515245 + 12345) \bmod 2^{15} \\
&= (X_2 * 1103515245 + 12345) \bmod 2^{15}
\end{aligned}
$$

Another Problem: Output is non-random

- Since each output of **`rand()`** depends only on previous output of **`rand()`**, guessing first value of rand gives you the key.
  - Keyspace is now reduced to 2^{15}.
  - Left as exercise: first byte of key is sufficient to derive all other bytes of key, so key space is really 2^8. (I remain skeptical of this.)
- Bottom line: This implementation of **`rand()`** is totally insecure for cryptographic purposes, no matter how seed is chosen.
- Fact: On some platforms this is really how **`rand()`** is implemented.

Historical Fun

In 1995, it was discovered that Netscape browsers generated SSL session keys using the time and process ID as seed. This was guessable, so all SSL sessions were breakable in minutes. In fact, Netscape web servers generated their long-term RSA keypair in the same way, which was even worse.

More Historical Fun

Soon after the Netscape flaw was discovered, someone noticed that the random number generator in Kerberos was similarly flawed and keys were guessable in seconds. In fact, it had been flawed for years, and no one had noticed until then. The code contained some functions that provide secure random number generation, but they inadvertently hadn't been used due to a breakdown in the revision control process.

Even More Historical Fun

Four years later, someone found a different flaw in the (supposedly fixed) Kerberos random generator: there was a misplaced memset() call that was intended to zero out the seed after it was used, but actually zeroed out the seed ***before*** it was used, ensuring that an all-zeros seed would be used to generate Kerberos keys.

More Historical Fun Than is Reasonable

Also in 1995, the XWindows ``magic cookie'' authentication method was discovered to have a serious flaw in how it generated magic cookies: it used rand() exactly as shown in the code snippet at the beginning of this lecture, and consequently there were only 28 possible magic cookies. It only took a fraction of a second to try them all and gain unauthorized access to someone else's X display.

OK, So There's Lots of Historical Fun

Around the same time, someone discovered that NFS (Network Filesystem) filehandles were predictable in Sun's NFS implementation. Sun used the time of day and process ID to seed a random number generator, and the filehandle was calculated from this seed. Also, in the NFS protocol, anyone who knows a valid filehandle can bypass the authentication protocol. This meant that anyone could defeat Sun's NFS security simply by guessing the seed and trying all corresponding filehandles.

More Recent Fun (Circa 2008)

Similar flaws have been found in DNS resolvers, which would allow an attacker to send spoofed DNS responses and have them accepted by vulnerable DNS clients.

More Recent Fun

Majordomo (a mailing list manager on Unix systems) used a bad random number generator when sending subscription confirmation messages, which would allow an attacker to subscribe some poor victim to thousands of mailing lists and forge confirmations that appear to come from the victim.

More Recent Fun

- At one point, someone noticed that PGP had been using the return value from read() to seed its pseudorandom number generator, rather than the contents of the buffer written by read(). Since read() always returned 1 (the number of bytes read), this meant that the seed was a stream of 1s, so session keys were predictable.

More Recent Fun

- Around the year 2000, a fun example came to light: one online poker site used an insecure pseudorandom number generator to shuffle the deck of cards. A player could see the cards in their own hands, derive some partial information about a few of the outputs from this pseudorandom number generator, and infer the seed used to shuffle the deck. This lets a smart player infer what cards everyone else holds, which obviously allows one to rake in the cash at the poker table. Oops. Fortunately, the folks who discovered the flaw notified the web site and wrote a paper rather than exploiting it to cheat others.

More Recent Fun

But we’ve fixed this all now.

More Recent Fun

- But we’ve fixed this all now. NOT

Most Recent Fun

- **PuTTY Private Key Recovery (2024 - CVE-2024-31497):** A critical vulnerability in the PuTTY SSH client (widely used) arose because the PRNG used for generating nonces (a number used once) in ECDSA signatures was seeded improperly. While the PRNG produced "random" numbers, it didn't have enough entropy, allowing researchers to predict the nonce, recover private keys, and hijack connections.
- **Weak Seed Usage in IoT/Secomea (2024 - CVE-2024-1579):** The Secomea GateManager (used for remote access) had a vulnerability due to improper seed usage in its PRNG. This allowed attackers to predict session IDs, leading to potential session hijacking.
- **DFINITY/Internet Computer Identity Weakness (2024):** A Snyk security advisory highlighted a "Use of Cryptographically Weak Pseudo-Random Number Generator" in a DFINITY identity library. It was caused by using a predictable seed for key pair generation, which could lead to key compromises.

Most Recent Fun

- **Widespread IoT/SSL Certificate Vulnerabilities (Ongoing/Recent Audit Findings):** Research has shown that many embedded devices and routers use weak PRNGs when generating RSA keys for TLS certificates. Because the seeds are often predictable (based on device uptime or low-entropy hardware sources), many certificates share factors, making them easy to break.
- **Predictable Session IDs (2023-2024):** Various web applications (including Objectplanet Opinio) have been flagged for using weak PRNGs for token generation. These tokens, which should be random to prevent hijacking, are instead generated based on predictable, time-based, or low-entropy seeds, allowing attackers to guess user session IDs.

The Underlying Problem:
These incidents usually fall under CWE-336 (Same Seed in PRNG) or CWE-337 (Predictable Seed in PRNG). Instead of using a *Cryptographically Secure* Pseudo-Random Number Generator (CSPRNG), developers use basic PRNGs—perfect for simulations but unsafe for security—resulting in guessable secrets.

Generating Pseudorandom Numbers

- *True random number generators* (TRNG): generates bits that are distributed uniformly at random, so that all outputs are equally likely, and with no patterns, correlations, etc.
- *Cryptographically secure pseudorandom number generator* (CS-PRNG). A CS-PRNG generates a sequence of bits that appear, as far as anyone can tell, to be indistinguishable from true random bits. CS-PRNGs use cryptographic techniques to achieve this task.

Typically Two Step Process

- Generate a seed.
  - Typically use a TRNG to generate a short seed that is truly random. The seed only needs to be long enough to prevent someone from guessing it.
    - E.g., the seed might be 128 bits. The seed plays a role similar to that of a cryptographic key.
  - Using a TRNG ensures that the seed will be unpredictable by any attacker.

Typically Two Step Process

- Generate pseudorandom output, using this seed.
  - CS-PRNG is used to stretch seed to a long pseudorandom output.
    - Modern cryptographic CS-PRNGs allow generation of essentially unlimited amount of output (billions of bits are no problem).
    - Using a CS-PRNG ensures that the pseudorandom bits thus generated have no discernible patterns.

Typically Two Step Process

- The cryptographic properties of the CS-PRNG ensure that using pseudorandom bits instead of true-random bits makes no (detectable) difference.

So, need to know:

- How to build a CS-PRNG
- How to build a TRNG

Cryptographyically Secure PRN Generation

- Is this even possible?
  - 128 bits amplified to billions?
  - It’s possible, especially with help of good cipher

Cryptographyically Secure PRN Generation

- One possibility
  - Think of seed as cryptographic key
  - Pick some symmetric key cipher (e.g., AES)
  - Encrypt fixed message using key and cipher
    - For many ciphers, ciphertext generated is indistinguishable from random bits

Cryptographyically Secure PRN Generation

- Ex. Generate n bits from 128-bit seed k by AES-CBC(k, 0n)
  - Proven a secure CS=PRNG if AES is a secure block cipher
    - Note that any IV will do here for CBC mode
  - Also efficient: need only one call to AES per 128 bits of output generated
    - We've already seen how fast we can generate output using AES.

Formalizing "Secure PRNG"

- Suppose PRNG that expands 128 bits to 1,000,000
  - PRNG is deterministic function $G:\{0,1\}^{128} \rightarrow \{0,1\}^{1,000,000}$ that maps seed s to output G(s)
- Let K denote random variable distributed uniformly on $\{0,1\}^{128}$
- Let U denote random variable distributed uniformly on $\{0,1\}^{1,000,000}$
- Roughly: G is secure if the output G(K) is indistinguishable from U
  - E.g., no attacker A can tell whether a sequence of 1,000,000 bits is generated by G or U.

Formalizing "Secure PRNG"

- More precisely: an attacker A is an algorithm that takes a sequence of one million bits as inputs, and outputs a guess as to how those bits computed.
- The *advantage* of the attacker A is given by

$$\text{Adv A} = \left|\Pr[A(G(K)) = \text{"pseudorandom"}] - \Pr[A(U) = \text{"pseudorandom"}]\right|$$

- G is CS-PRNG is there is no feasible attacker A that has non-negligible advantage at breaking G
- Really no more to this: any good crypto library will provide implementation of CS-PRNG

Blum Blum Schub Pseudorandom Number Generation

- Security is provably reduced to difficulty of factoring large semiprimes
- The setup (similarities to RSA)
  - N = pq, the product of two large primes
  - x_0, the seed, which must be a large number that is co-prime to p and q (p and q are not factors of it) and not 0 or 1.
- Generate the sequence:
  - $x_{i+1} = x_i^2 \bmod N$
  - Pseudorandom sequence is generated by taking the least significant bit of each x_i

Blum Blum Schub Pseudorandom Number Generation

- Generate the sequence:
  - $x_{i+1} = x_i^2 \bmod N$
  - Pseudorandom sequence is generated by taking the least significant bit of each xi
- x_0, p, and q must be kept secret
- Properties: Given any portion of the sequence, computationally infeasible to learn what came before or what comes after.
- Security reduces to: factoring is at least as difficult as another hard problem: Quadratic residuosity

True Random Number Generation

- Building a TRNG is more challenging.
  - On some platforms, CPU provides built-in hardware capability to generate truly random bits using an appropriate random physical process.
  - Unfortunately most platforms do not have this
- Another possibility: hardware peripheral that acts as a dedicated TRNG, again using truly random physical processes.
  - Cost between a few dollars and thousands, depending on how fast bits generated, and desired security level

So what do we do if there is no perfect source of physically random bits?

In that case we're hosed. You can't generate randomness out of nothing.

Anything that Alice can compute with a deterministic algorithm, the attacker can compute, too.

However…

- Often have some sources of randomness, though these are typically imperfect.
  - E.g, the source might generate values that are somewhat unpredictable, but are not uniformly distributed.
  - Or, some of the bits may be predictable by adversary.
  - Some sources may be slightly unreliable: they have some probability of failure, where a failed source emits completely predictable outputs (e.g., all zeros).

Examples

- A high-speed clock.
  - Some machines have clock with nanosecond precision. If an adversary can only predict the time on your machine to within a microsecond (because of clock skew), then the low 10 bits are unpredictable.
    - Windows: millisecond granularity
    - Mac: microsecond granularity

Examples

- A sound card.
  - Thermal noise will cause some randomness in samples from a microphone input with nothing plugged in.
    - These samples not uniformly distributed (e.g., they may contain 60Hz hum from line noise), but they are not totally predictable, either.

How Do We Use These?

- One source on its own is probably insufficient.
  - Clock might contribute 10 bits of randomness, sound card maybe 30 bits, keyboard maybe 30 bits, and disk timings maybe 20 bits (to make up some numbers).
    - In total, this provides 90 bits of randomness, assuming the sources are independent, which ought to be more than enough in the aggregate.

How Do We Use These?

- So, combine data, perhaps by concatenation
  - This is enough to ensure that adversary cannot guess exact value of concatenation
    - But concatenation will not be uniformly distributed. In general, entropy might be spread out among values in strange and unpredictable ways.

Standard Solution

- Use a cryptographic hash function, such as SHA3.
  - Seem to have the property that they do a good job of extracting uniformly-distributed randomness from imperfect random sources.
    - Ex. Suppose that x is a value from an imperfect source, or concatenation from multiple sources. Suppose also it is impossible for adversary to predict entire value x, except with negligible success probability (say, with probability $1/2^{160}$). Then SHA3(x) is a 256-bit value whose distribution is (we think) approximately the uniform distribution.
  - Generally, if it is impossible to predict exact value of x except with probability $1/2^n$, and if SHA3 is secure, then truncating SHA3(x) to n bits should provide an n-bit value that is uniformly distributed.

In Practice

- Built exactly as described:
  - Identify as many sources of randomness as you can
  - Collect samples from each source
  - Concatenate sampled values
  - Hash them with a secure hash
  - Truncate to appropriate length
- Result is a short true-random value that can be used as a seed for a CS-PRNG

Caution

- Some bad sources
  - IP addresses: adversary will likely know this
  - Content of network packets: assume an adversary can see these
    - Network packet timings bad too, since adversary may be able to predict these depending on precision of timer
  - Process IDs
    - Process IDs for many servers and daemons are easily predictable, since boot process is deterministic and thus anything started at boot time is likely to received the same pid each time the machine is booted. Assume the adversary knows our OS, so for many processes the pid doesn't contribute any useful randomness.

In General

- Want values that will be random and unpredictable even to an adversary.
- Also, because cost of RNG failure is so high, we are usually very conservative when we analyze potential randomness sources, and we evaluate them according to the assumptions that are most favorable to the attacker (among all scenarios that are remotely plausible).