CS 161 Computer Security
Fall 2005 Joseph/Tygar/Vazirani/Wagner **Notes 21**

> "Anyone who uses software to produce random numbers is in a state of sin."
> —John von Neumann

> "The generation of random numbers is too important to be left to chance."
> —Robert R. Coveyou

Random Number Generation

Many cryptographic protocols require the parties to generate random numbers. For instance, cryptographic keys have to be generated in a way that makes them unpredictable to anyone other than the authorized creator of the key. How should we generate these random numbers?

In this lecture, you will learn two things: how to generate truly random bits; and cryptographic methods to stretch a little bit of true randomness into a large stream of pseudorandom values that are indistinguishable from true random bits.

1 What Can Go Wrong

It turns out that random number generation is very easy to get wrong. For instance, what is wrong with the following code?

```
unsigned char key[16];

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

There all sorts of problems with this code. Can you spot any of them?

In case you are not familiar with the `rand()` function, here is a quick refresher. Here are their function prototypes:

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

Each call to `rand()` returns a pseudorandom value in the range `0` to `RAND_MAX`, calculated as a deterministic function of the seed. Also, `srand(s)` sets the seed to `s`. For instance, here is one possible implementation of `rand()` and `srand()`:

```
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;
}
```

Finally, `time(NULL)` returns the current time, as measured by the number of seconds since Jan 1, 1970.

With that background, here are two of the security holes in the code snippet listed previously:

- It is easy to guess the value of the key. The seed is highly predictable, and anyone who can guess the seed can calculate the value of the cryptographic key. The algorithm that `rand()` uses for computing its outputs as a function of the seed is publicly known.

  Consequently, anyone who guess the time at which the key was generated can apply the very same algorithm to infer each of the bytes of the key. For instance, if Alice generates a new session key at the start of each session using this code, then anyone who eavesdrops on a session will probably be able to determine the time of day on Alice's machine at the start of the session (and hence the seed passed to `srand()`) and then decrypt everything that is encrypted using this session key.

  Even if the eavesdropper doesn't know at what time the key was generated, there just aren't that many possibilities. For instance, suppose we know the key was generated this year. There are $3600 \times 24 \times 365 = 31,536,000 \approx 2^{25}$ seconds in a year. This means that, if I can narrow down when the key was generated to within a one-year window, then there are only 2^{25} possible values for the seed passed to `srand()`. Of course, it is not hard to try each one of them, and see which key would result in each case. Also, once you have found the right key, it is easy to recognize this fact: you can simply try decrypting the ciphertext with each potential key, and see which one provides plausible-looking cleartext (the wrong keys will cause the decryption to look like gibberish; checksums and message authentication codes will be invalid; and so on). Since we have narrowed down the key to only 2^{25} possibilities, the system essentially has 25 bits of strength. Modern machines can try all 2^{25} possibilities in seconds or minutes, which demonstrates that cracking this system is utter triviality.

- The output of this generator isn't very random. If we use the sample generator listed above, then the sequence of 16 key bytes will be highly non-random. For instance, it turns out that the low bit of each successive output of `rand()` will alternate (e.g., $0,1,0,1,0,1,\ldots$). Do you see why? The low bit of `x * 1103515245` is the same as the low bit of `x`, and then adding `12345` just flips the low bit. Thus the low bit alternates. This narrows down the set of possible keys to only 2^{113} possibilities; much less than the desired value of 2^{128}.

  Actually, it's even worse than that: each output from `rand()` depends only on the last output from `rand()`. Do you see why? If we let $N_0, N_1, N_2, \ldots$ denote the sequence of values of `next` during successive calls to `rand()`, then on a 32-bit machine we have $N_{i+1} = 1103515245 \times N_i + 12345 \bmod 2^{32}$. Moreover if we denote the output from the ith call to `rand()` by X_i, we have $X_i = N_i \bmod 2^{15}$. From this it follows that $X_{i+1} \equiv 1103515245 \times X_i + 12345 \pmod{2^{15}}$. In other words, each output from `rand()` depends only on the previous output from `rand()`. If I can guess the first output from `rand()`, then I will be able to derive all the subsequent outputs from `rand()`. This means that there are only 2^{15} possibilities for the key. In fact, there are only 2^8 possibilities for the key, as the first byte of the key is sufficient to derive all the other bytes (left as an exercise). Thus this implementation of `rand()` is totally insecure, for cryptographic purposes, and it would remain insecure no matter how we generate the seed.

By the way, it might be hard to believe, but on some platforms this really is how `rand()` is implemented. Truly. No joke.

These might look like mistakes that only a bonehead could make, but they have appeared in real code. Here are some examples of systems that have been been found vulnerable, due to poor random number generation:

- 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.
- 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 up till 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.
- 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.
- Also in 1995, the X Windows "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 2^8 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.
- 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.
- 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.
- Majordomo 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.
- 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 recently, 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 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.

The lesson to take away is that these flaws are rampant, and their consequences can be severe.

These flaws suggest that there are two essential principles for secure random number generation:

- *Seeds must be unpredictable.* There should be sufficiently many possibilities that they cannot be enumerated (2^{128} is enough). All possibilities should be equally likely, and nothing the attacker knows should help him to predict the value of the seed. Ideally, the seed would be truly random.
- *The algorithm for generating pseudorandom bits must be secure.* There should be no discernable patterns in the pseudorandom bits produced by this algorithm. Even if an attacker manages to learn (or guess) some of the pseudorandom bits, this should not help him predict the value of any other pseudorandom bits that he does not already know.

We will see next how these principles may be met.

2 Generating Pseudorandom Numbers: Outline

We can distinguish two different kinds of random number generation:

- *True random number generator (TRNG).* A 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. Typically, there is some short true-random seed, which is then stretched in a deterministic way to a long output in a way that ensures that the output is computationally indistinguishable from true-random. CS-PRNGs use cryptographic techniques to achieve this task.

As you might have guessed by now, in most systems the random number generation process is structured as follow:

1. *Generate a seed.* We 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. For instance, 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.
2. *Generate pseudorandom output, using this seed.* We then use a CS-PRNG to stretch the seed to a long pseudorandom output. Modern cryptographic CS-PRNGs allow to generate an essentially unlimited amount of output (billions of bits are no problem). Using a CS-PRNG ensures that the pseudorandom bits thus generated have no discernable patterns.

Then, any place in the system that would normally require a random value can instead use pseudorandom bits computed by the CS-PRNG. The cryptographic properties of the CS-PRNG ensure that using pseudorandom bits instead of true-random bits makes no (detectable) difference.

The rest of this lecture is broken down into two parts: (1) how to build a CS-PRNG; (2) how to build a TRNG.

3 Cryptographically Secure Pseudorandom Number Generation

The function of a CS-PRNG might seem magical. Given only 128 bits of randomness, a CS-PRNG is supposed to somehow amplify it into billions of random bits. How on earth can that be possible? Surprisingly, it turns out that this is not only possible, but it is quite easy, once you know a little cryptography.

One very simple idea is to think of the seed as a cryptographic key, pick some symmetric-key encryption scheme, encrypt some fixed message under this key, and use the ciphertext as our pseudorandom bits. For many encryption schemes, the ciphertext is indistinguishable from true-random bits. For instance, we might generate n pseudorandom bits from a 128-bit seed k as AES-CBC$(k, \mathbf{0}^n)$, where $\mathbf{0}^n$ denotes the all-zeros message that is n bits long[1]. It can be proven that this is a secure CS-PRNG, assuming that AES is a secure block cipher. This CS-PRNG is also very efficient, because it needs only one call to AES per 128 bits of output generated. Standard AES implementations can generate output at rates in excess of 50 MB/sec on a modern desktop machine, so this is extremely efficient.

We can formalize the requirement for what it means for a CS-PRNG to be secure. Suppose we have a CS-PRNG that expands a 128-bit seed to one million bits of output. This CS-PRNG is a deterministic function $G : \{0,1\}^{128} \to \{0,1\}^{1,000,000}$ that maps the seed s to the output $G(s)$. Let K denote a random variable that is distributed uniformly at random on $\{0,1\}^{128}$, and let U denote a random variable distributed uniformly on $\{0,1\}^{1,000,000}$. Roughly speaking, we say that G is secure if the output $G(K)$ is indistinguishable from U. In other words, no attacker A who is given a sequence of one million bits can tell whether this sequence was contains pseudorandom bits computed by applying G to a random 128-bit seed, or contains true-random bits generated by one million coin flips. More precisely, an attacker A is an algorithm that takes a sequence of one million bits as input, and outputs a guess as to how those bits were computed. The advantage of the attacker A at breaking G is given by

$$\mathrm{Adv}\, A = |\Pr[A(G(K)) = \text{“pseudorandom”}] - \Pr[A(U) = \text{“pseudorandom”}].$$

The function G is a secure CS-PRNG if there is no feasible attacker A that has a non-negligible advantage at breaking G.

There is essentially nothing more to say about CS-PRNGs. They are easy to build, and any good cryptographic library will provide an implementation of a CS-PRNG.

4 True Random Number Generation

Building a TRNG is more challenging. On some platforms, it is easy to build a TRNG, because the CPU provides built-in hardware capability to generate truly random bits by using an appropriate random physical process. However, unfortunately most platforms do not have this capability.

Another possibility is to buy a hardware peripheral that acts as a dedicated TRNG, against using truly random physical processes. These can be obtained for only a few hundred dollars, so for high-value servers, this might be a reasonable solution.

However, for many systems—including desktop machines—this is not viable. We have to make do with what we've got. How do we build a TRNG when there is no perfect source of physically random bits?

[1] We can use any fixed IV, such as the all-zeros IV.

Obviously if there are no sources of randomness available, we're hosed. One cannot generate randomness out of nothing. Anything that Alice can compute with a deterministic algorithm, the attacker can do, too.

However, often we do have some sources of randomness available, though these sources are typically imperfect. For instance, the source might generate values that are somewhat unpredictable, but are not uniformly distributed. Or, some of the bits may be predictable by an adversary. Also, some sources may be slightly unreliable: they might have some probability of failure, where a failed source emits completely predictable outputs (e.g., all zeros). Some examples:

- A high-speed clock. Some machines have a 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.
- A soundcard. If we sample from a microphone input with nothing plugged in, then thermal noise will cause some randomness in the samples. These samples are not uniformly distributed (for instance, they may contain 60Hz hum from line noise), but they are not totally predictable, either.
- Keyboard input. PGP asks the user to type some keys randomly during key generation, and uses the keys typed—as well as the time between each pair of key presses—as a randomness source. This is an imperfect source, as not all keys are equally likely (uppercase letters are probably less common than lowercase letters, for instance).
- Disk timings. Seek latencies on disks vary in a random random, due to air turbulence inside the disk. The random variability is very slight, and disk access times are highly correlated, but there is some empirical evidence that it may be possible to extract on the order of random bit per second.

There are many more possible sources, but these are representative.

How do we take advantage of these sources to build a TRNG? Notice that any one source on its own is probably insufficient. The clock might contribute only 10 bits of randomness, the soundcard might contribute another 30 bits, and the keyboard might provide 30 bits, and disk timings might provide another 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.

This suggests we ought to combine data from many sources. But how do we do it? The obvious first step is to concatenate all the values sampled from each of the sources. This is enough to ensure that an adversary cannot guess the exact value of this concatenation, but the concatenation will not be uniformly distributed. In general, the entropy might be spread out among these values in funny ways. For instance, a list of one hundred disk timings might have 20 bits of entropy in all, but spread across in funny ways, with strange and hard-to-characterize correlations.

The standard solution is to use a cryptographic hash function, such as SHA1. Cryptographic hash functions seem to have the property that they do a good job of extracting uniformly-distributed randomness from imperfect random sources. Suppose that x is a value from an imperfect source, or a concatenation of values from multiple sources. Suppose moreover that is impossible for an adversary to predict the entire value x, except with negligible success probability (say, with probability $\leq 1/2^{160}$). Then $\text{SHA1}(x)$ is a 160-bit value whose distribution is, we think, approximately the uniform distribution. More generally, if it is impossible to predict the exact value of x except with probability $1/2^n$, and if SHA1 is secure, then truncating $\text{SHA1}(x)$ to n bits should provide a n-bit value that is uniformly distributed.

And this is exactly how people often build TRNGs in practice. You identify as many sources of randomness as you can; you collect samples from each source; you concatenate all those sampled values; you hash them with SHA1; and (if necessary) you truncate to an appropriate length. The result is a short true-random value, which can be used as a seed for a CS-PRNG.

4.1 Caution

It is worth pointing out that some sources that might seem attractive are actually useless for cryptography. Some examples:

- IP addresses. Useless, since we'd better assume that an adversary will know our IP address and thus be able to predict this value.
- Contents of network packets. We can't count on this to have any predictability, since an eavesdropper can see all of these values. Network packet timings may also be predictable, too, depending on the precision of our timer.
- Process IDs. The process IDs for many servers and daemons are easily predictable, since the boot process is deterministic and thus anything started at boot time is likely to received the same pid each time the machine is booted. We'd better assume that the adversary knows what operating system we are running, so for many processes the pid doesn't contribute any useful randomness.

In general, we want values that will be random and unpredictable even to an adversary. Also, because the cost of a 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).

5 Practical Considerations

Many programming environments provide utility functions that implement all of this for you, so you don't have to. Many Unix platforms provide a special device `/dev/urandom`; reading from it provides pseudorandom bytes. On Microsoft platforms, you can use `CryptGenRandom()`. In Java, you can use `java.security.SecureRandom`. All of these generate pseudorandom bytes by applying a CS-PRNG to a short seed collected with a TRNG. Many cryptographic libraries also provide APIs with this functionality: see, e.g., `RAND_bytes()` in the open-source OpenSSL library.

Avoid `rand()`, `random()`, `drand48()`, `java.util.Random`, Mersenne twister, LSFRs, LCRNGs, and any other PRNG that hasn't been verified to be cryptographically secure.