CS5431 – Computer Security Practicum	Spring 2017
Lecture 5: Key Management	
March 3, 2017	*Instructor: Eleanor Birrell*

1 Key Management

So far, you have seen an overview of the basic cryptographic protocols that are used to secure a variety of modern systems. In 5430, you talked about symmetric encryption (e.g., AES), asymmetric encryption (e.g., RSA, El Gamal), hybrid encryption, MACs, digital signatures, and key agreement protocols. This means you are now familiar with many of the essential cryptographic building blocks that are used to design secure systems.

However, successfully security a system using cryptographic tools inherently presumes successful *key management*. Key management is the problem of managing cryptographic keys in a system; this includes key generation, storage, use, and replacement. A failure in any of these areas can introduce a vulnerability that might result in compromised security.

1.1 Key Generation

There are two primary decisions to be made regarding key generation. First, what keys should you generate? You need to decide on an algorithm and (if applicable) a key length. And second, where are you going to find the randomness needed to generate those keys?

To answer the first question, we turn to the National Institute of Standards and Technology (NIST), a unit of the United States Commerce Department responsible for maintaining and promoting measurements and standards. NIST gives recommendations for appropriate algorithms and key sizes depending on the function and the security function of the key and the security life of the data. These recommendations are summarized in Table 1. Recommendations are given in terms of *bits of security*, logically speaking a cryptographic scheme with n bits of security means that an adversary needs to perform 2^n operations to break it.

Security (in bits)	Symmetric Enc.	RSA	Elliptic Curve Crypto	HMAC	NIST Rec.
≤ 80	2TDEA	k = 1024	f = 160-223		No
112	3TDEA	k = 2048	f = 224-255		until 2030
128	AES-128	k = 3072	f = 256-383	SHA-1	Yes
≥ 256	AES-256	k = 15360	f = 512+	SHA-256	Yes

Table 1: NIST Key Length Recommendations

Failure to generate the right sorts of keys can result in embarrassing vulnerabilities.

After Yahoo! revealed that attackers had stolen account information for more than a billion accounts in August 2013, it also revealed that it had stored passwords for those accounts using the outdated MD5 hashing algorithm. It is assumed that most of those passwords were therefore successfully recovered by the attackers, and all users were advised to change their passwords for their Yahoo account and for any other accounts that used the same password.

To answer the second question, we consider the available resources. In Unix-type systems, there are two special files that serve as sources of pseudorandomness: `/dev/random` and `/dev/urandom`. `/dev/random` is a blocking pseudorandom number generator (PRG) that collects environmental noise from device drivers and other sources and generates pseudorandom bits. Historically, Linux systems used a hash-based PRG, but since 2014 most current implementations have used a stream cipher called chacha20; macOS and iOS continue to use a SHA-1 based PRG. If there a process requests more bits than are currently available in `/dev/random`, the call blocks until more entropy has been collected. `/dev/urandom` is an "unlimited" nonblocking pseudorandomnumber generator that uses stream ciphers to produce as much randomness as needed; there are no known attacks based on systems using `/dev/urandom` instead of `/dev/random`. In Java, this means you should use SecureRandom (on Linux systems it generally returns random bytes from directly from `/dev/random` or `/dev/urandom` depending on the system defaults and the input parameters. You should not use Java.util.Random for cryptographic key generation; Java.util.Random is implemented as a linear congruential pseudorandom number generator with a 48-bit seed and does not provide enough entropy.

As we discussed last week, insufficient entropy during key generation can undermine the security of your system and render in vulnerable to a variety of attacks.

1.2 Key Storage

There are two approved techniques for storing cryptographic keys. Keys can either be stored remotely or in a cryptographic module.

Remote key storage is appropriate for high-value keys that are not frequently needed. The remote storage location is assumed to be secure (for example, because it is air-gapped). In many cases, remotely stored key are stored as *secret shares*. Secret shares are logically pieces of the key with the property that an n out of the k shares can be used to reconstruct the original key. Such a scheme provides redundancy—it is not vulnerable to the loss or corruption of a single share—and defense in depth—the adversary must successfully compromise n shares in order to reconstruct the key. One simple secret-sharing protocol encodes shares as points on a polynomial and the secret as the value $f(0)$; a $(n-1)$-degree polynomial is uniquely defined by any n points, so a n out of k secret sharing can be achieved by defining k points on such a polynomial.

The alternative technique, which is useful for keys that are used more frequently—is to store the keys in a cryptographic module. Intuitively, this just means that plaintext keys are only available in memory; an adversary who does not have access to memory can-

Key Type	Cryptoperiod	
	Originator (OUP)	Recipient
Private Signature Key	1 to 3 years	-
Public Signature-Verification Key	Several years (depends on key size)	
Symmetric Authentication Key	$\leq$ 2 years	$\leq$ OUP + 3 years
Private Authentication Key	1 to 2 years	
Public Authentication Key	1 to 2 years	
Symmetric Data Encryption Keys	$\leq$ 2 years	$\leq$ OUP + 3 years
Symmetric Key Wrapping Key	$\leq$ 2 years	$\leq$ OUP + 3 years
Symmetric Master Key	About 1 year	
Private Key Transport Key	$\leq$ 2 years	
Public Key Transport Key	1 to 2 years	
Symmetric Authorization Key	$\leq$ 2 years	
Private Authorization Key	$\leq$ 2 years	
Public Authorization Key	$\leq$ 2 years	

Table 2: NIST Cryptoperiod Recommendations

not retrieve the keys. Java provides a variety of cryptographic module implementations as Java Keystores.

1.3 Key Use and Replacement

In general, a single key should only be used for a single purpose. For example, the same key should never be used for both encryption and signing. This limits the damage if a key is compromised, precludes the possibility of key leakage from the other protocol, and enables prompt key destruction when appropriate. Plaintext keys stored on disk or in memory should be zeroed out after use (not just deallocated)

Keys should only be used for a limited period of time, known as a *cryptoperiod*. This practice limits the time and data available for cryptanalysis, limits the exposure if a single key is compromised, and enables prompt adoption of new algorithms when appropriate. The appropriate length for a cryptoperiod depends on the strength of the cryptographic mechanism, the operating environment, the security life of the data, the security function (e.g., encryption, signing, key protection), the key update process, and the threat model. Cryptoperiods are typically shorter for encrypting communications than for encrypting stored date, due to the overhead of re-encryption.

2 Alternative Cryptographic Approaches

A system that relies on standard public-key (asymmetric) encryption, private-key (symmetric) encryption, or hybrid encryption is inherently going to have to deal with the problem of key management. This, unfortunately, includes a lot of opportunities to make mistakes and to thereby introduce vulnerabilities. Modern cryptographers have therefore developed techniques for simplifying key management.

2.1 Password-Based Encryption

Password-based encryption (PBE) grew out of the observation that key management could be simplified if the secret keys didn't have to be stored anywhere because users just remembered them. Of course, it is not reasonable to expect your users to remember large strings of random bytes (or large random primes), but it is (perhaps) reasonable to assume that users can remember passwords. The idea behind password-based encryption is basically to derive encryption keys from user passwords. A naive password-based encryption algorithm could use the password as the seed for a pseudorandom number generator or hash function and use the output as the secret key.

Of course, there are problems with using passwords to generate keys. Most notably, password are less random than randomly chosen encryption keys. Last time, we discussed some of the vulnerabilities that arise when RSA keys are generated with insufficient randomness; our naive password-based encryption protocol would inherently be vulnerable to this type of attack. But how bad would it be in practice? Well, a uniformly generated AES key has 128 bits of entropy. A 2048-bit RSA key has 112 bits of entropy. A uniformly generated 8-character password composed of keyboard characters has about 52 bits of entropy. But users don't choose their passwords uniformly at random. For example, SplashData's annual survey of leaked passwords indicates that the top five passwords in 2016 were (1) 123456, (2) password, (3) 12345, (4) 12345678, and (5) football. In fact, according to NIST estimates, a typical user-chosen 8-character password has 18-24 bits of entropy, depending on whether the user is required to include particular types of characters. This is low enough to make our naive password-based encryption protocol vulnerable not only to the types of insecure-randomness attacks we discussed last week but also to guessing or *dictionary attacks*.

There are two general approaches to mitigating this vulnerability in our naive password-based encryption scheme. The first technique is to make computation slow by using repeated iterations of the pseudorandom number generator (generally implemented with a secure hash function such as SHA-256). Current practice uses about 10,000 iterations. NIST recommends a minimum of 1000 iterations and up to 10 million iterations for security-critical applications where performance is not critical. However, slowing down the computation of a key does not prevent an adversary from compiling a pre-computed "dictionary" of keys derived from common passwords.

The second approach is to augment the key with random bytes known as a *salt*.

The salt is randomly generated anew for each value encrypted. The random bytes are combined with the password to increase the entropy of the derived key; the salt is then stored alongside the ciphertext so that it can be decrypted in the future. Note that in order to be effective, the salt must be chosen with high-quality randomness. In Java, this means you should use SecureRandom (on Linux systems it generally draws directly from the native PRG, that is it returns random bytes from /dev/random (blocking) or /dev/urandom (nonblocking, uses a stream cipher), both of which generate bits from environmental noise collected from devise drivers and other sources), not Java.util.Random (which is implemented as a linear congruential pseudorandom number generator with a 48-bit seed).

One common way to combine these techniques into a secure password-based encryption scheme—standardized as PKCS12—is as follows: (1) select a fresh salt value uniformly at random, (2) append the password to the salt and also append a counter value, initialized to 1, (3) calculate a secure hash of the concatenated value, (4) repeat steps (2)-(3) with the output for the prescribed number of iterations, incrementing the counter with each iteration. A variant of this protocol designed to produce variable-length keys—PKCS5v2, sometimes referred to as PBKDF2—is approach is believed to offer the best available security for password-based encryption. A schematic for PBKDF2 is shown in Figure 1. Note that Java provides built in password-based encryption ciphers; however, these implementations rely on insecure underlying algorithms. Stronger ciphers are supported by Bouncy Castle.

Password-based encryption is commonly used in cases where a single user wants to secure the confidentiality of data stored for later retrieval. This is commercially available as a variety of full-disk encryption tools (e.g., FileVault, Bitlocker, TrueCrypt), and more recently incorporated directly into a variety of niche cloud storage services (e.g., SpiderOak, Wuala, Tresorit, Mega). A variant of PBE designed for establishing secure connections, known as password authenticated connection establishment (PACE) is currently in development. A protocol is specified in RFC 6631, but it is not an Internet Standards Track specification.

2.2 ID-based Encryption

A standard problem with encryption is how to solve the key distribution problem. That is, how do users discover the secret key associated with a principal. On the Internet, this problem is solved by deploying *certificate authorities*, hierarchies of trusted authorities that attest to certificate chains containing public keys. However, this scheme assumes both that the intended recipient is online when the communication occurs and that the PKI infrastructure will scale to support the number of authenticated principals. If a system design calls for sending encrypted messages directly to a user—instead of only using keys to authenticate servers—these assumptions might not hold.

The idea behind ID-based encryption (IBE) is that in many applications, each user has an identifier (a username, an email address, etc) that is known to the principals that

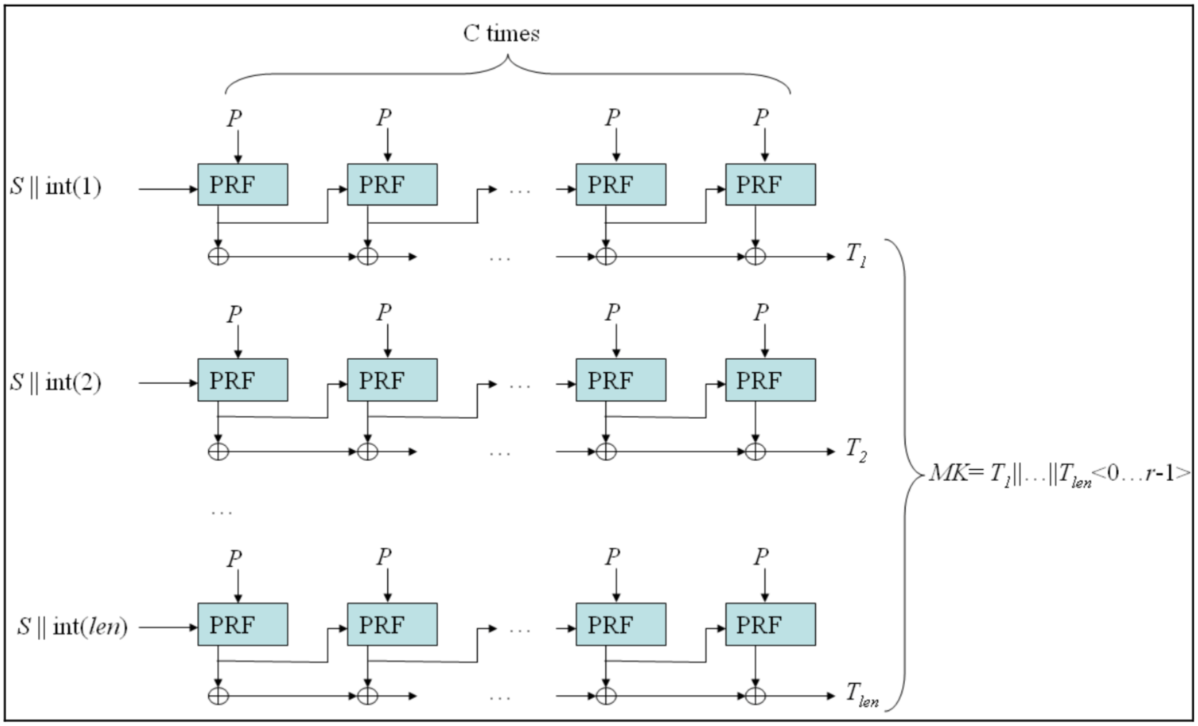

Figure 1: Schematic of the PBKDF2 password-based encryption protocol.

want to send messages to that user. ID-based encryption simply uses this identifier as the public key associated with that user.

Existing IBE protocols work with the aide of a highly-trusted principal called the *key generator*. The key generator generates a master public key and a master private key. The master public key is published and used to encrypt ciphertexts under an identifier. The master private key is used to derive individual private keys for each user; to learn their private key, each user must contact the key generator and prove their identity.

Feasible IBE protocols have been known since 2001. A preliminary IBE standard is specified in RFC 5091; it is not an accepted Internet Standard. However, IBE protocols tend to have higher overhead than standard encryption techniques, and the trust model is not always appropriate for particular systems; IBE is not generally deployed in existing commercial products.

Generalizations of ID-based encryption have also been proposed, most notably attribute-based encryption (ABE). ABE is a form a public-key encryption in which the secret key of a user is dependent on attributes associated with that user (e.g., role in the system, country of residence). ID-based encryption can be viewed as a special case of ABE in which the identifier is the only attribute each user has. ABE schemes have be proposed by cryptographic researchers, but none are sufficiently efficient to have been standardized or deployed in commercial products.