Most Common Cryptography Mistakes

4/2/2014
#1: Don’t Roll Your Own

- Don’t design your own crypto algorithm
- Use a time-honored, well-tested system—e.g., SSL, PGP, SSH
#2: Don’t Encrypt without Auth

- Common mistake: encrypt, but no authentication
  - A checksum does not provide authentication
- If you’re encrypting, you probably want authenticated encryption
  - Encrypt-then-authenticate: $E_{k_1}(M), F_{k_2}(E_{k_1}(M))$
  - Or, use a dedicated AE mode: GCM, EAX, ...
#3: Be Careful with Randomness

- Common mistake: use predictable random number generator (e.g., to generate keys)
- Solution: Use a crypto-quality PRNG.
  - /dev/urandom, CryptGenRandom, ...
char chall[16], k[16];

srand(getpid() + time(NULL) + getppid());
for (int i=0; i<16; i++)
    chall[i] = rand();
for (int i=0; i<16; i++)
    chal[i] = rand();
where \((R, K) = \text{hash(microseconds, x)}\)

\[x = \text{seconds} + \text{pid} + (\text{ppid} \ll 12)\]
where \((R, K) = \text{hash(microseconds, } x\text{)}\)
\[
x = \text{seconds} + \text{pid} + (\text{ppid} \ll 12)
\]

Attack: Eavesdropper can guess \(x\) (\(\approx 10\) bits) and microseconds (20 bits), and use \(R\) to check guess.
Bad PRNGs = broken crypto

- Netscape server’s private keys ($\approx 32$ bits)
- Kerberos v4’s session keys ($\approx 20$ bits)
- X11 MIT-MAGIC-COOKIE1 (8 bits)
- Linux vtun ($\approx 1$ bit)
- PlanetPoker site ($\approx 18$ bits)
- CryptoAG – NSA spiked their PRNG
#4: Passphrases Make Poor Keys

- Common mistake: Generate crypto key as Hash(passphrase)
- Problem: \( \approx 20 \) bits of entropy; even with a slow hash, this is not nearly enough. Human-generated secrets just don’t have enough entropy.
- Solution: Crypto keys should be random.
#5: Be Secure By Default

- Common mistake: Security is optional, or configurable, or negotiable
- Fix: There is one mode of operation, and it is secure. No human configuration needed.
  - e.g., Skype
Wardriving / Access Point Mapping

468 WEP
1,265 Clear
1,733 Total

Pasadena Networks, LLC http://www.pasadena.net
MS Point-to-Point Encryption (MPPE)

If both endpoints support 128-bit crypto:

\[ M \oplus RC4(K) \]

where \( K = \text{hash(password || R)} \)
MS Point-to-Point Encryption (MPPE)

If both endpoints support 128-bit crypto:

- I support 128-bit crypto
- So do I. Here’s a nonce: R
- $M \oplus RC4(K)$

where $K = \text{hash(password || R)}$

Attack 1: Eavesdropper can try dictionary search on password, given some known plaintext.
MS Point-to-Point Encryption (MPPE)

If both endpoints support 128-bit crypto:

I support 128-bit crypto

So do I. Here’s a nonce: R

\[ \text{M} \oplus \text{RC4(K)} \]

where \( K = \text{hash(password } || | R) \)

Attack 2: Active attacker can tamper with packets by flipping bits, since there is no MAC.
I support 128-bit crypto
So do I. Here’s a nonce: R
M ⊕ RC4(K)

where K = hash(password || R)

I support 128-bit crypto
So do I. Here’s a nonce: R
M ⊕ RC4(K)

Attack 3: Bad guy can replay a prior session, since client doesn’t contribute a nonce.
I support 128-bit crypto

So do I. Here’s a nonce: $R$

$M \oplus \text{RC4}(K)$

where $K = \text{hash(password} \mid \mid R)$

---

**Attack 4:** Bad guy can replay and reverse message direction, since same key used in both directions.
MS Point-to-Point Encryption (MPPE)

If one endpoint doesn’t support 128-bit crypto:

\[
M \oplus \text{RC4}(K)
\]

where \(K = \text{hash(uppercase(password))}\)
MS Point-to-Point Encryption (MPPE)

If one endpoint doesn’t support 128-bit crypto:

Client

I support 128-bit crypto

I don’t. Use 40-bit crypto

Server

M ⊕ RC4(K)

where K = hash(uppercase(password))

Attack 1: Eavesdropper can try dictionary search on password, given some known plaintext.
MS Point-to-Point Encryption (MPPE)

If one endpoint doesn’t support 128-bit crypto:

- **Client**: I support 128-bit crypto
- **Server**: I don’t. Use 40-bit crypto

\[ M \oplus RC4(K) \]

where \( K = \text{hash(uppercase(password))} \)

**Attack 2**: Dictionary search can be sped up with precomputed table (given known plaintext).
MS Point-to-Point Encryption (MPPE)

Client

I support 128-bit crypto

I don’t. Use 40-bit crypto

Bad Guy

$M \oplus \text{RC4}(K)$

where $K = \text{hash(uppercase(password))}$

Attack 3: Imposter server can downgrade client to
40-bit crypto, then crack password.
MS Point-to-Point Encryption (MPPE)

where $K = \text{hash(uppercase(password))}$, $K' = \text{hash(password || R)}$

**Attack 4:** Man-in-the-middle can downgrade crypto strength even if both client + server support 128-bit crypto, then crack password.
#6: Careful with Concatenation

- Common mistake: Hash(S || T)
  - “builtin” || “securely” = “built” || “insecurely”
Amazon Web Services

http://amazon.com/set?u=daw&n=David&t=U&m=...

MAC(K,″udawnDavidtU″)
Amazon Web Services

://amazon.com/set?u=daw&n=DavidtAq&t=U&m=...

MAC(K,"udawnDavidtAqtU")

://amazon.com/set?u=daw&n=David&t=A&qt=U&m=...
#6: Careful with Concatenation

- Common mistake: Hash(S || T)
  - “builtin” || “securely” = “built” || “insecurely”
- Fix: Hash(len(S) || S || T)
- Make sure inputs to hash/MAC are uniquely decodable
#7: Don’t re-use nonces/IVs

- Re-using a nonce or IV leads to catastrophic security failure.
Credit card numbers in a database

<table>
<thead>
<tr>
<th>dgaTkyuPS8bs4rPXoQn3</th>
</tr>
</thead>
<tbody>
<tr>
<td>dgaalSeET8Hv4rvfpQrz</td>
</tr>
<tr>
<td>cQGakyuFQcri6brfoAH6Jg==</td>
</tr>
<tr>
<td>dgWdmSuESSro4bfXpQj0</td>
</tr>
<tr>
<td>cQSYmCKLScDt4bDXqAj2Ig==</td>
</tr>
<tr>
<td>cQWTlCKNSsfr5bDfqAnzIw==</td>
</tr>
<tr>
<td>cAKdkyOMT8Ti6LvQpwj2IA==</td>
</tr>
</tbody>
</table>
After Base64 decoding

| 76 06 93 93 2b 8f 4b c6 ec e2 b3 d7 a1 09 f7 |
| 76 06 9a 95 27 84 4f c1 ef e2 bb df a5 0a f3 |
| 71 01 9a 93 2b 85 41 ca e2 e9 ba df a0 01 fa 26 |
| 76 05 9d 99 2b 84 4a ca e8 e1 b7 d7 a5 08 f4 |
| 71 04 98 98 22 8b 49 c0 ed e1 b0 d7 a8 08 f6 22 |
| 71 05 93 94 22 8d 4a c7 eb e5 b0 df a8 09 f3 23 |
| 70 02 9d 93 23 8c 4f c4 e2 e8 bb d0 a7 08 f6 20 |