Passwords, RNGs, Implementation issues

CS642: Computer Security

Professor Ristenpart

http://www.cs.wisc.edu/~rist/

rist at cs dot wisc dot edu
More topics in crypto

- Password-based crypto
- Password cracking, WPA
- Random number generators (RNGs)
- Side channel attacks
Symmetric encryption

Correctness: \( D( K, E(K,M,R) ) = M \) with probability 1 over randomness used
Password-based symmetric encryption

Correctness: \( D(K, E(K, M, R)) = M \) with probability 1 over randomness used
Ciphertext is $C,T$

How do we use with a pw?
Password-based Key Derivation (PBKDF)

PBKDF(pw,salt):

\[ \text{H} \]

\[ \text{H} \]

\[ \text{...} \]

\[ \text{H} \]

\[ \text{K1} \]

\[ \text{H} \]

\[ \text{K2} \]

\[ \text{Truncate if needed} \]

\[ \text{pw} \text{|| salt} \text{|| 1} \]

\[ \text{repeat c times} \]
PBKDF + Symmetric encryption = PW-based encryption

**Enc(pw,M,R):**
- salt || R’ = R
- K = PBKDF(pw,salt)
- C = Enc’(K,M,R’)
- Return (salt,C)

Here Enc’ is a normal symmetric encryption scheme (CBC-HMAC)

**Dec(pw,C):**
- salt || C’ = C
- K = PBKDF(pw,salt)
- M = Enc’(K,C’)
- Return M

What can go wrong?
**Recommendation:** It should not be a name, a slang word, or any word in the dictionary. It should not include any part of your name or your e-mail address.

Almost all of the 5000 most popular passwords, that are used by a share of 20% of the users, were just that – names, slang words, dictionary words or trivial passwords (consecutive digits, adjacent keyboard keys, and so on). The most common password among Rockyou.com account owners is “123456”. The runner up is “12345”. The following table depicts the top 20 common passwords in the database list:

<table>
<thead>
<tr>
<th>Rank</th>
<th>Password</th>
<th>Number of Users with Password (absolute)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>123456</td>
<td>290731</td>
</tr>
<tr>
<td>2</td>
<td>12345</td>
<td>79078</td>
</tr>
<tr>
<td>3</td>
<td>123456789</td>
<td>76790</td>
</tr>
<tr>
<td>4</td>
<td>Password</td>
<td>61958</td>
</tr>
<tr>
<td>5</td>
<td>iloveyou</td>
<td>51622</td>
</tr>
<tr>
<td>6</td>
<td>princess</td>
<td>35231</td>
</tr>
<tr>
<td>7</td>
<td>rockyou</td>
<td>22588</td>
</tr>
<tr>
<td>8</td>
<td>1234567</td>
<td>21726</td>
</tr>
<tr>
<td>9</td>
<td>12345678</td>
<td>20553</td>
</tr>
<tr>
<td>10</td>
<td>abc123</td>
<td>17542</td>
</tr>
<tr>
<td>11</td>
<td>Nicole</td>
<td>17168</td>
</tr>
<tr>
<td>12</td>
<td>Daniel</td>
<td>16409</td>
</tr>
<tr>
<td>13</td>
<td>babygirl</td>
<td>16094</td>
</tr>
<tr>
<td>14</td>
<td>monkey</td>
<td>15294</td>
</tr>
<tr>
<td>15</td>
<td>Jessica</td>
<td>15162</td>
</tr>
<tr>
<td>16</td>
<td>Lovely</td>
<td>14950</td>
</tr>
<tr>
<td>17</td>
<td>michael</td>
<td>14898</td>
</tr>
<tr>
<td>18</td>
<td>Ashley</td>
<td>14329</td>
</tr>
<tr>
<td>19</td>
<td>654321</td>
<td>13984</td>
</tr>
<tr>
<td>20</td>
<td>Qwerty</td>
<td>13856</td>
</tr>
</tbody>
</table>

If a hacker would have used the list of the top 5000 passwords as a dictionary for brute force attack on Rockyou.com users, it would take only one attempt (per account) to guess 0.9% of the users passwords or a rate of one success per 111 attempts. Assuming an attacker with a DSL connection of 55KBPS upload rate and that each attempt is 0.5KB in size, it means that the attacker can have 110 attempts per second. At this rate, a hacker will gain access to one new account every second or just less than 17 minutes to compromise 1000 accounts. And the problem is exponential. After the first wave of attacks, it would only take 116 attempts per account to compromise 5% of the accounts, 683 attempts to compromise 10% of accounts and about 5000 attempts to compromise 20% of accounts. The following diagram depicts the expected effectiveness of attacks using a small, carefully chosen, attack dictionary:

From an Imperva study of released RockMe.com password database
Brute-force attacks

• Given known plaintext, ciphertext pair:
  – M and C = Enc(pw,M)

• Enumerate a dictionary D of possible passwords

\[
\text{BruteForce1}(M,C): \\
\text{foreach } pw^* \text{ in } D \text{ do} \\
\quad C^* = Enc(pw^*,M,R) \\
\quad \text{If } C^* = C \text{ then} \\
\quad \text{Return } pw^*
\]

Both are public:
C = salt || IV || C1 || ...

R is salt || IV in CBC-based modes

\[\text{IV} \quad \text{E}_{K1} \quad \text{C0} \quad \text{C1}\]
Brute-force attacks

• Given known plaintext, ciphertext pair:
  – M and C = Enc(pw,M)
• Enumerate a dictionary D of possible passwords

\[
\text{BruteForce1}(M,C):
\text{foreach pw* in D do}
\hspace{1cm} C^* = Enc(pw^*,M,R)
\hspace{1cm} \text{If } C^* = C \text{ then}
\hspace{2cm} \text{Return pw*}
\]

\[
\text{BruteForce2}(C):
\text{foreach pw* in D do}
\hspace{1cm} M^* = Dec(pw^*,C)
\hspace{1cm} \text{If } M^* \text{ looks right then}
\hspace{2cm} \text{Return (pw*,M*)}
\]
PBKDF design attempts to slow down brute-force attacks

Iterating \( c \) times should slow down attacks by factor of \( c \)

Salts:
- Different derived keys, even if same password
- Slows down attacks against multiple users
- Prevents precomputation attacks, if salts chosen correctly

\[
pw \ || \ salt \ || \ 1 \rightarrow H \rightarrow H \rightarrow \ldots \rightarrow H \rightarrow K1
\]
Say $c = 4096$. Generous back of envelope* suggests that in 1 second, can test 252 passwords and a naïve brute-force:

<table>
<thead>
<tr>
<th>Password Complexity</th>
<th>$P = 10^6$</th>
<th>Time Estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 numerical digits</td>
<td>$10^6 = 1,000,000$</td>
<td>$\sim 3968$ seconds</td>
</tr>
<tr>
<td>6 lower case alphanumeric digits</td>
<td>$36^6 = 2,176,782,336$</td>
<td>$\sim 99$ days</td>
</tr>
<tr>
<td>8 alphanumeric + 10 special symbols</td>
<td>$72^8 = 722,204,136,308,736$</td>
<td>$\sim 33$ million days</td>
</tr>
</tbody>
</table>

* I did the arithmetic...
802.11 WPA passwords

PMK = PBKDF( pw, ssid | ssidlength ) with c = 4096

PTK = H( PMK || ANonce || SNonce || AP MAC address || STA MAC address )

MIC = HMAC-MD5(PTK, 2\textsuperscript{nd} message)

So after sniffing one handshake by another party, we can mount offline brute force attack
802.11 WPA passwords

PMK = PBKDF( pw, ssid | ssidlength )
with c = 4096

PTK = H( PMK || ANonce || SNonce || AP MAC address || STA MAC address )

MIC = HMAC-MD5(PTK, 2^{nd} message)

BruteForce2(MIC,ANonce,SNonce,2^{nd} message):

foreach pw* in D do

PMK* = PBKDF(pw*,ssid | ssidlength)

PTK* = H(PMK* || ANonce || ... )

MIC* = HMAC-MD5(PTK*, 2^{nd} message)

If MIC* = MIC then
Return pw*
We can also use precomputation for common SSID’s

\[ PMK = F(pw, ssid) \]

\[ MIC = G(PMK, data) \]

\[ PMK = \text{PBKDF}(pw, ssid | | ssidlength) \text{ with } c = 4096 \]

\[ PTK = H( PMK | | A\text{N}once | | S\text{N}once | | AP \text{ MAC address} | | \text{STA MAC address}) \]

\[ MIC = \text{HMAC-MD5}(PTK, 2^{nd} \text{ message}) \]

**Online(\(D, SsidList\))**:

- **for each** \(pw^*\) in \(D\) do
  - **for each** \(ssid^*\) in \(SsidList\) do
    - \(PMK^* = F(pw^*, ssid^*)\)
    - \(T[PMK^*] = pw^*\)
    - Add \(PMK^*\) to \(P[ssid^*]\)

Return \(P, T\)

**Online(\(P, T, MIC, A\text{N}once, ...\))**:

- **for each** \(PMK^*\) in \(P[ssid]\) do
  - \(MIC^* = G(PMK^*, data)\)
  - If \(MIC^* = MIC\) then
    - Return \(T[PMK^*]\)

Time-space trade-off
Password recap

• Short passwords can be cracked easily (JohnTheRipper, aircrack)
• Salting and iteration help
  – Salts must be sufficiently large and unpredictable

From xkcd.com
• Random number generation
• Measure events on system, harvest entropy (unpredictability from them)
  – keyboard presses and timing
  – file/network interrupts
  – mouse movements
• Hash entropy down to “extract” (hopefully) uniform bit strings
Linux /dev/random

Linux random number generator (2500 lines of undocumented code)

Diagram from [Gutterman, Pinkas, Reinman 2006]

Applications like TLS take **randomness** from /dev/random

They then maintain an internal pool of **random bits**
MD_Update(&m,buf,j);
....
MD_Update(&m,buf,j); /* purify complains */

These lines of code commented out from OpenSSL random number generator code (md_rand.c) to address complaints by security tools Purify and Valgrind

Only the PID was used as input to RNG.

It took a ~2 years for the bug to be (publicly) discovered!

Debian OpenSSL bug lead to small set of possible
Debian Bug Leaves Private SSL/SSH Keys Guessable

Posted by timothy on Tuesday May 13 2008, @12:01PM
from the security-is-a-process dept.

SecurityBob writes

"Debian package maintainers tend to very often modify the source code of the package they are maintaining so that it better fits into the distribution itself. However, most of the time, their changes are not sent back to upstream for validation, which might cause some tension between upstream developers and Debian packagers. Today, a critical security advisory has been released: a Debian packager modified the source code of OpenSSL back in 2006 so as to remove the seeding of OpenSSL random number generator, which in turns makes cryptographic key material generated on a Debian system guessable. The solution? Upgrade OpenSSL and re-generate all your SSH and SSL keys. This problem not only affects Debian, but also all its derivatives, such as Ubuntu."

Reader RichiH also points to Debian's announcement and Ubuntu's announcement.
“Protect Against Adware and Spyware: Users protect their PCs against adware, spyware and other malware while browsing the Internet with Firefox in a virtual machine.”
[http://www.vmware.com/company/news/releases/player.html]

“Your dad can do his [private] surfing on the virtual machine and can even set it to reset itself whenever the virtual computer is restarted, so there's no need to worry about leaving tracks. ... I recommend VMware because you can download a free version of VMware Server for home use.”
"Protect Against Adware and Spyware: Users protect their PCs against adware, spyware and other malware while browsing the Internet with Firefox in a virtual machine."

[http://www.vmware.com/company/news/releases/player.html]
Virtual machine resets lead to RNG failures

To-be-used randomness captured in snapshot!

Recent versions of Firefox, Chrome allow session compromise attacks

Apache mod_ssl TLS server: server’s secret DSA key can be stolen!

[R., Yilek – NDSS ‘10]
User launches browser in VM

Randomness gathered by browser random number generator (RNG)

User snapshots VM

Snapshot later run.

User requests https page

Randomness used by TLS key transport

A second run from snapshot leads to same secret key being sent to (different) server

https://www.mybank.com/

TLS session key transport

A logical timeline of events
RNG recap

• Randomness is often a weak link in crypto implementations
• Building a good RNG is not always easy
• Intel RNG instructions in next generation chips
Side-channel attacks

• Implementations might leak information about secret internal state via side-channels:
  – power consumption
  – Electromagnetic emanations (Tempest)
  – timing
  – Shared physical resources (CPU cache)
PKCS #1 RSA encryption

Kg outputs \((N,e),(N,d)\) where \(|N|_8 = n\)

Let \(B = \{0,1\}^8 / \{00\}\) be set of all bytes except 00

Want to encrypt messages of length \(|M|_8 = m\)

**Enc**((\(N,e\), \(M\), \(R\))

\[\text{pad} = \text{first } n - m - 2 \text{ bytes from } R \text{ that are in } B\]

\[X = 00 || 02 || \text{pad} || 00 || M\]

Return \(X^e \text{ mod } N\)

**Dec**((\(N,d\), \(C\))

\[X = C^d \text{ mod } N \quad ; \quad aa || bb || w = X\]

If \((aa \neq 00)\) or \((bb \neq 02)\) or \((00 \notin w)\)

Return error

\[\text{pad} || 00 || M = w\]

Return \(M\)
Textbook exponentiation

**ModExp**(X,e,N)

X’ = X
For i = 2 to d do
    X’ = X’*X mod N
Return X’

**SqrAndMulExp**(X,e,N)

b_k,...,b_0 = e
f = 1
For i = k down to 0 do
    f = f^2 mod N
    If b_i = 1 then
        f = f*X mod N
Return f
SqrAndMulExp(X,e,N)

\[ b_k, ..., b_0 = e \]
\[ f = 1 \]
For i = k down to 0 do
  f = f^2 \mod N
  If \( b_i = 1 \) then
    f = f^X \mod N
Return f

\[ e = \sum_{b_i \neq 0} 2^i \]
\[ X^e = X^{\sum_{b_i \neq 0} 2^i} = \prod_{b_i \neq 0} X^{2^i} \]

\[ X^e \mod N = \left( \prod_{b_i \neq 0} (X^{2^i} \mod N) \right) \mod N \]

\[ X^{11} = x^{1+2+8} = (x)(x^2)(x^8) \]
\(\text{SqrAndMulExp}(X,e,N)\)
\(b_k, \ldots, b_0 = e\)
\(f = 1\)

For \(i = k\) down to 0 do
  \(f = f^2 \mod N\)
  If \(b_i = 1\) then
    \(f = f \times X \mod N\)

Return \(f\)

But:
Squaring and multiplying take different amounts of time and power.

From Messerges et al. 1999:

**Fig. 2. Cross-Correlation of Multiplication and Exponentiation Power Signals**
The above signals were obtained using the power analysis equipment described in Section 4.
SqrAndMulExp(X, e, N)

\[ b_k, ..., b_0 = e \]
\[ f = 1 \]

For \( i = k \) down to 0 do
   \[ f = f^2 \mod N \]
   If \( b_i = 1 \) then
       \[ f = f \times X \mod N \]

Return \( f \)

But:
Squaring and multiplying take different amounts of time and power.

Remote timing attacks against other (Boneh, Brumley 2003)
Chosen ciphertexts + timing = key extraction
\sim 1 \text{ million queries (though highly variable)}
Lots of other implementation pitfalls

- Hard-coded keys in binaries
- Default passwords
- Developing your own crypto algorithms
- Poor key management (Kerberos, RADIUS)