The Book of Bad Crypto Decisions (part 1 of 1,000,000)

Orr Dunkelman

Computer Science Department
University of Haifa

21th July, 2014

אוניברסיטת חיפה
University of Haifa

Outline

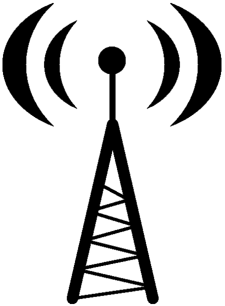
1. The Motivation for this Talk
2. The (In)Security of GSM
   - Quick Introduction to how GSM Works
   - And One Key to Rule Them All
   - How (Not) to Authenticate Control
   - Compress-Encrypt-Authenticate-Error Correct
3. On the Importance of Entropy in Security
   - Why Entropy is Needed
   - How to Generate Entropy
   - The Debian Bug
   - How (Not) to Select Prime Numbers
4. Whose Key is it Anyway?
   - How (Not) to Build a PKI
   - The (Not) so Subtle Way to Validate Certificates

The Motivation for this Talk

- There are many **small** design decisions with **huge** impact on security.
  - Things which make sense from efficiency point of view, but completely destroy security.
  - Things which are counter-intuitive ("but why would it hurt security?")
  - Things that used to work one way, but the world has changed...
  - Common (and not so common) mistakes.

Welcome to the World of GSM/3G

- The most widely deployed mobile phone technology.
- More than 3G users around 212 countries.
- Has inherent support for roaming.
- GSM uses 4 bands: 900MHz/1800MHz in most of the world, and 850MHz/1900MHz in North America and Chile.
- 3G uses the 1700/2100 MHz band.



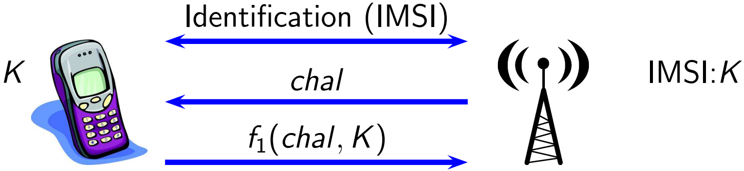
Security of GSM/3G

- Mobile phones are susceptible to many threats:
  1. Call theft
  2. Cell phone duplication
  3. Eavesdropping
  4. . . .
- To deal with them, GSM/3G incorporate several security mechanisms, which are based on a (table of) preshared secret embedded into the SIM card.

Security of GSM/3G (cont.)

- To handle the authentication of the mobile phone, a pair of protocols are executed: A3/A8.
- The two protocols perform authentication and key exchange, based on the preshared secret.
- At the end of A3/A8 execution the mobile phone and the operator have a session key of 64 bits (or 128 bits in 3G).
- A3/A8 is not specified in the standards, but many operators decided to deploy COMP128, which proved a bad decision as COMP128 is extremely weak [GW98].
- Today, most operators run secure algorithms, such as COMP128v2.

A3/A8 — General Structure



The session key is set as $f_2(chal, K)$.

Session keys and A5/1 and A5/2

- The 64-bit session key is used to key A5/1 (or A5/2).
- Each phone needs to support both ciphers (and today, also A5/3 and A5/4).
- The cipher to be used is selected by the network (export control/support at basestation).

What happens when the cipher is changed?

Changing a Cipher — The Easy Solution

- The session key is secret.
- Deriving a new key requires executing A3/A8 again.
- This actually should never happen...
- Easy solution: use same key for A5/1, A5/2, etc. (not A5/4).

What could possibly go wrong?

Quick and Dirty Introduction to A5/1

- 64-bit key stream cipher.
- Uses 3 LFSRs of lengths 19,22,23.
- LFSRs are loaded with the key and a frame number.
- Then the are irregularly clocked.
- Best attack (before disclosure): 2^{48} time (for a little data).
- Best attack (after disclosure): 2^{40} time (more data).
- Conclusion: not the best option, but decent enough.

Quick and Dirty Introduction to A5/2

- 64-bit key stream cipher.
- Uses 4 LFSRs of lengths 17,19,22,23.
- LFSRs are loaded with the key and a frame number.
- Then the 17-bit register controls the clocking.
- Given the contents of the 17-bit register, breaking the system is trivial (everything becomes linear).
- Original attack: 2^{17} trials (each taking a bit). About a second of computation.
- Best attack: Precompute 2^{17} inversion matrices. Find key with a simple matrix multiplication.
- Conclusion: weak. very weak.

Can you see the problem?

How to Attack GSM

- Start your own basestation.
- Stand close to the cell phone you are attacking.
- Ask the cell to use your basestation (over an unencrypted control channel).
- Ask the cell to talk A5/2 with you.
- Break A5/2.
- Remove your basestation.
- Let target switch back to A5/1 or A5/3.
- Make profit.

You can allegedly buy devices that do all this work for you.

Conclusions (and Mitigation)

- Each context should has its own keys.
- In the theory of cryptography this is called "domain separation".
- Main reason: another layer of defense (breaking part of the system does not violate the full security).
- Additional reason: helps in the debug (though you need to debug the different contexts).
- In the case of GSM, could have been session key is output of $f_2(chal, K, alg)$ (where alg is A5/1 or A5/2 or A5/3).

Unauthenticated Control Channel

Recall the active attack on GSM:

- Start your own basestation.
- **Start your own basestation.**
- Stand close to the cell phone you are attacking.
- Ask the cell to use your basestation (over an unencrypted control channel).
- **Ask the cell to use your basestation (over an unencrypted control channel).**
- Ask the cell to talk A5/2 with you.
- **Ask the cell to talk A5/2 with you.**
- Break A5/2.
- Remove your basestation.
- Let target switch back to A5/1 or A5/3.
- Make profit.

Unauthenticated Control Channel (cont.)

- Control channel can also tell the cell to switch off encryption completely (A5/0).
- But then, the adversary just hears what he is forwarding.
- Protection of control data is important (not just due to this attack).
- Allows meta data to leak (control channel lets you start phone calls).
- It should be hidden (protecting privacy of users) and authenticated (authenticating both ways).
- Helps in preventing rouge basestations.
- Similar attacks are also applicable to TOR (the onion routing network).

Conclusions (and Mitigation)

- Encrypt & authenticate all channels.
- Can be done using encryption (preferably under a different key than the session key).
- Authenticate identity basestations (i.e., two-way authentication).
- Can be done in the first message (basestation sends $f_3(chal, K)$).
- Main reasons:
  - Security (another layer of defense),
  - Privacy,
  - Prevents active attacks.

How to Attack A5/2 Efficiently

- As mentioned earlier A5/2 is a stream cipher.
- Once a 17-bit register is known, the entire algorithm becomes linear.
- A simple straightforward attack — guess the 17 bits, and break a linear scheme.
- A more advanced attack — precompute the matrices that "break" the linear scheme.
- But this requires multiplying a vector with a matrix 2^{17} times.
- And actually, requires knowing some conversation bits.

Is there a better way?

How to Attack A5/2 Efficiently (cont.)

- Luckily, in GSM the following procedure is used in the encryption:
  - Take the message M
  - Apply error correction code (very expanding) $ECC(M)$
  - Encrypt with A5/2 $ECC(M) \oplus KS$
- Recall that KS is actually one of 2^{17} linear functions $L_i(X)$ (for a 64-bit internal state X).
- In other words, the ciphertext is $ECC(M) \oplus L_i(X)$.

How to Attack A5/2 Efficiently (cont.)

- Both ECC and L_i are expanding linear operations.
- In other words, it is easy to compute a kernel of "a joint" matrix $ECC \oplus L_i$, which operates on M and X.
- Attack:
  - For all $ECC \oplus L_i$, compute the kernel of the matrix.
  - Given ciphertext-only, see in which kernel it is found.
  - It will be in one kernel...
- Once L_i is found, game is over.

Conclusions (and Mitigation)

- Thou shalt not do anything besides the following order:
    - Compression
    - Encryption
    - Authentication (MAC)
    - Error correction
- Use authenticated encryption when possible.
- For public-key scenarios, consider signcryption (or sign and then encrypt).

Randomness

> *Randomness means lack of pattern or predictability in events.*

[Wikipedia]

- Randomness offers many great difficulties for us on an every day base.
- Luckily for us, it has also great security uses.

The Bright Side of Randomness

- If no one cannot predict the future, then so does the adversary.
- Which means that when you select cryptographic keys, you should probably pick random keys (to reduce chance of being guessed).
- Just like when selecting passwords — the smaller the entropy of the password, the easier it is to guess it.

How to Generate Entropy (in Hardware)

> *Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.*

[John von Neumann, 1951]

How to Generate Entropy (in Hardware)

- Random bit (number) generation in hardware relies on various physical traits:
    - Nuclear decay,
    - Real dices,
    - Complex (chaotic) systems (e.g., lava lamps),
    - Sampling a circuit with an odd number of not gates,
    - ...
- Some of these methods do produce equally distributed stream of bitsbut they are correlated.
- Usually involves a (cryptographic) post-processing to handle correlation.
- Check FIPS 140-2 concerning evaluation of the quality of the produced randomness.

How to Generate Entropy (in Software)

- You cannot.
- Software (without bugs) is completely predictable.
- The system may have some physical sources of randomness (entropy):
    - Hard-disk access times
    - Network activity
    - User interface (keyboard/mouse/. . .)
    - Process id
    - Leftovers in memory
    - New on Intel platforms: RDRAND

How to Use Entropy (Software)

- /dev/random (TRNG) vs. /dev/urandom (seed that goes into a PRNG).
- When generating keys — ONLY /dev/random.
- And post-process.
- And try to combine with other sources of entropy.
- And try to use a hardware RNG.

The Debian Bug — OpenSSL

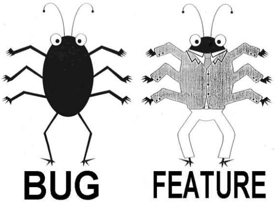
- OpenSSL is the most common open source cryptographic suite (implements SSL/TLS).
- It handles its own key generation, on top of the /dev/random offered by the system.
- In September 2006, a Debian developer (*kroeckx*) commented out the following line:

  MD_Update(&m,buf,j);

- (Actually, he commented this line twice).
- The reason: Valgrind complained about using an uninitialized data structure — buf.

The Debian Bug — OpenSSL (cont.)

- One problem — buf contained some “random” leftovers.
- Without it, the only “randomness” the PRNG of OpenSSL was seeded with was the process id.
- One of $2^{15} = 32768$ possible values. . .



Impact

- If there are only 32,768 seeds, there are at most 32,768 different random sequences that may be produced.
- Even in the key generation phase of OpenSSL (and of OpenSSH).
- Meaning: whoever produced a public key between 2006 and the discovery (2008), used low-entropy keys.
- Which can be factored, reversed (signatures), etc.
- Lots and lots of affected systems. Including small network devices.

Conclusions (and Mitigation)

- If it ain't no broken, don't fix it, eh?
- Randomness: diversify your sources.
- Randomness: more sources cannot hurt you (unless there are hidden correlations).
- Run randomness tests.
- Test for randomness — in the stream, and across streams (would have identified WEP attacks as well).
- Remember: You can only **FAIL** at randomness tests.

Selecting Prime Numbers

1. Pick a random seed.
2. Put into a PRNG.
3. Produce a stream of bits.
4. Take a chunk of bits, and test whether they compose a random number.
5. If so, output number. If more random primes are needed, go to Step 3.
6. If the number is not prime, go to Step 3.

What can Possibly Go Wrong?

1. Pick a random seed.
2. **Put into a PRNG**.
3. **Produce a stream of bits**.
4. Take a chunk of bits, and test whether they compose a random number.
5. If so, output number. If more random primes are needed, go to Step 3.
6. If the number is not prime, go to Step 3.

In Theory there is no Difference between Theory and Practice . . .

- [H+12] gathered 12.8M TLS public keys and 10.2M SSH public keys.
- Using some quick algorithms (DJB's algorithm) they found pairs of keys that share prime numbers.
- Such pairs of keys allow using $gcd(\cdot)$ to find the prime numbers themselves (i.e., factorizing the RSA key)
- Which is a bad thing. . .

Summary of [H+12] Results

	TLS	SSH
Total # of Keys	12,828,613	10,216,363
Repeated Keys (RKs)	7,770,232 (60.5%)	6,642,222 (65.0%)
Vulnerable RK	714,243 (5.57%)	981,166 (9.6%)
Default Keys	670,391 (5.23%)	
Low-entropy RK	43,852 (0.34%)	
Factored RSA keys	64,081 (0.5%)	2,459 (0.03%)
Compromised DSA keys		105,728 (1.03%)
Debian weak keys (!)	4,147 (0.03%)	53,141 (0.52%)
512-bit RSA keys	123,038 (0.96%)	8,459 (0.08%)

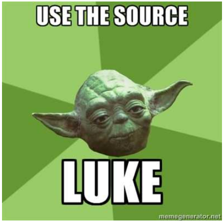
What Went Wrong? (partial list)

- Sites using default keys. with certificates(!)
- Citrix servers using shared keys (again some with certificates).
- Most repeated keys — ok (used in hosting services). Some — low entropy of the PRNG.
- Many routers, server management cards, VPN devices, VoIP products, and network storage devices suffered from these issues.

But Why?

Recall the Entropy Sources:

- Hard-disk access times — SSDs do not have as diverse access times.
- Network activity — Network devices are initialized in quiet networks.
- User interface (keyboard/mouse/...) — most devices no longer have a lot of user interface.
- Process id — system starts assigning pids at 0.
- Leftovers in memory — No leftovers — devices have a "zeroed" memory.



Conclusions (and Mitigation)

- OS developers:
    - Expose good randomness.
    - Explicitly define randomness assumptions.
- Library developers:
    - Set default at most secure option (OpenSSL used /dev/urandom).
    - Do not generate keys immediately one after the other (let some entropy "brew").
    - Pass OS information onwards.
- Developers:
    - Generate keys when needed (not in install/first boot).
    - Collect entropy.
    - NO DEFAULT KEYS!
    - Consider seeding entropy a-priori at production.
    - Obey OS restrictions.

The Story of Flame

- Along with Stuxnet considered to be one of the worms used to hack Iranian nuclear effort.
- A very complicated and advanced malware.
- Probably installed by an infected USB device.

Wait!

How come it installed when software signatures are used?

Signing Code

- Generally speaking, today's code is digitally signed by authors.
- A digital signature $sig = S(M)$, is a string of bits that authenticate the source of a message M (including that it was not tampered with).
- To verify a signature, the recipient obtains the signer's public key pk and checks whether (sig, M) is valid according to pk.
- But how does the recipient know pk?

Quick and Dirty Introduction to Certificates

- Assume we have a trusted third party.
- OK, not 100% trusted with everything. Just that it is trusted enough to link identity with a public key.
- Then, this entity can sign "attestations" of the form (id, pk) saying user id has public key pk.
- Signature to be done using the trusted entity public key pk_{CA}.

How do we know pk_{CA}?

Quick and Dirty Introduction to Certificates (cont.)

- The idea: you know pk_{CA} in advance.
- In each browser there are about 100 pre-approved CAs (certification authorities), with their public key.
- What to do with a new CA?
- CAs are allowed to issue a "special" certificate of the form: $(newCA, id(newCA), pk(newCA))$.
- So if you know one of the CAs that signed a certificate for the new CA, you are set to go.
- Of course, you may not know any CA signing for the new CA. But maybe one of them has a certificate issued by a CA you do know...
- And this is called certificate chain.

What Failed in the Case of Flame?

- Flame was signed by **Microsoft**.
- But this is due to some cryptanalytic attack based on MD5 weaknesses.
- Roughly speaking, you never sign a message, but its digest.
- MD5 was well known since 2004 to be weak.
- Was still used in 2008.
- And the phasing out is still ongoing.
- But this is not what I am going to discuss.

What Failed in the Case of Flame? (cont.)

- The original certificate was issued to an (unknown) entity for use in Outlook systems.
- Due to mishandling of permissions on certificates, such certificates whose root was Microsoft, were allowed to **sign** code.
- Due to mishandling of permissions on certificates, such certificates whose root was Microsoft, were allowed to **sign** code.
- And the code was trusted because it was “approved” by Microsoft.
- In other words — you could install without user’s interaction/approval.

Conclusions (and Mitigation)

- Mitigate weak crypto.
- Do not allow installation without user's interaction (unless signed directly or with a special key).
- Trust is not transitive.
- Domain separation. Good for certificates (as well).

How to Validate a Certificate

- As mentioned before, each user has a list of trusted CAs:
    - Verisign,
    - Comodo,
    - Entrust,
    - . . .
    - CNNIC
- When validating a certificate, we check whether the signing key is known (and trusted).
- If not, we recursively validate the signing key.

CNIC — The Chinese are After You

- In 2010 the Chinese CNNIC was added to the list of trusted CAs of Firefox.
- In other words, any Firefox trusts certificates issued by CNNIC.
- Including for gmail. Or bankofamerica.com.
- In other words, a CA can issue certificates “incorrectly”.
- Partial solution: Check that the certificate was issued by someone related.

Diginator — The Iranian are After You

- In 2011, the Dutch CA, diginator was taken over by the Dutch government.
- Apparently, their systems were hacked.
- And their private key was used to sign rouge certificates for several domains (mostly google related).
- These certificates were used to spy on Iranian activists.
- After the forensics, diginator was shut down.

The Real Issue

- Obviously, joining the CA roots or hacking into a CA invalidates the entire security model.
- However, there are better attack vectors:
  - Users accept all certificates (self-signed, expired, etc.)
  - Users can be easily tricked to not use secure connections.
  - Users . . .
- But also developers are to blame. . .

The Real Issue (cont.)

- Not all applications check certificates.
- In [F+13] it was found out that:
  - Of about 13,500 applications in google play, only 17 implemented certificate validation correctly.
  - Common errors:
    - Accept all certificates (89%)
    - Only check expiration (7.5%)
    - Break SSL

The Reason

- Apparently, developers use self-signed certificate for tests.
- These certificates cause issues when using default implementations.
- So they google the error code. The first answer is "Set handle-validation-fails to null"
- Obviously, this is a good way to solve debugging issues.
- And ruin security if you do not handle validation errors after development ends.

Conclusions (and Mitigation)

- **Trust is not transitive.**
- Stress-test using real certificates.
- Implement certificate pinning.
- Ask google, think on your own (TM).
- Try to rely on libraries (and good ones).
- Or develop one. . .

Questions?

Thank you very much for your attention!