Certificates, Password Hashing, and Case Studies

CS 161 Fall 2022 - Lecture 10

Last Time: Public-Key Encryption and Digital Signatures

- Public-key cryptography: Two keys; one undoes the other
- Public-key encryption: One key encrypts, the other decrypts
  - Security properties similar to symmetric encryption
  - ElGamal: Based on Diffie-Hellman
    - The public key is g^b, and C_1 is g^r.
    - Not IND-CPA secure on its own
  - RSA: Produce a pair e and d such that $M^{ed} = M \bmod N$
    - Not IND-CPA secure on its own
- Hybrid encryption: Encrypt a symmetric key, and use the symmetric key to encrypt the message
- Digital signatures: Integrity and authenticity for asymmetric schemes
  - RSA: Same as RSA encryption, but encrypt the hash with the private key

Today

- Certificates: How do we distribute public keys securely?
- Password Hashing: How do we securely store passwords?
- Case studies
  - How can we discover bad cryptography?
  - How do we securely send messages in practice?
  - How can we report abusive behavior?

Certificates

Textbook Chapter 13

Review: Public-Key Cryptography

- Public-key cryptography is great! We can communicate securely without a shared secret
  - Public-key encryption: Everybody encrypts with the public key, but only the owner of the private key can decrypt
  - Digital signatures: Only the owner of the private key can sign, but everybody can verify with the public key
- What's the catch?

Problem: Distributing Public Keys

- Public-key cryptography alone is not secure against man-in-the-middle attacks
- Scenario
  - Alice wants to send a message to Bob
  - Alice asks Bob for his public key
  - Bob sends his public key to Alice
  - Alice encrypts her message with Bob’s public key and sends it to Bob
- What can Mallory do?
  - Replace Bob's public key with Mallory’s public key
  - Now Alice has encrypted the message with Mallory’s public key, and Mallory can read it!

Problem: Distributing Public Keys

Alice

Mallory

Bob

Generate PK_B, SK_B

Send PK_B

Send PK_M

Receive PK_M

Send $\{M\}_{PK_M}$

Decrypt $\{M\}_{PK_M}$

Send $\{M\}_{PK_B}$

Decrypt $\{M\}_{PK_B}$

Problem: Distributing Public Keys

- Idea: Sign Bob's public key to prevent tampering
- Problem
  - If Bob signs his public key, we need his public key to verify the signature
  - But Bob's public key is what we were trying to verify in the first place!
  - Circular problem: Alice can never trust any public key she receives
- You cannot gain trust if you trust nothing. You need a root of trust!
  - **Trust anchor**: Someone that we implicitly trust
  - From our trust anchor, we can begin to trust others

Trust-on-First-Use

- **Trust-on-first-use**: The first time you communicate, trust the public key that is used and warn the user if it changes in the future
  - Used in SSH and a couple other protocols
  - Idea: Attacks aren't frequent, so assume that you aren't being attacked the first time communicate
  - Also known as "**Leap of Faith**"

Certificates

- **Certificate**: A signed endorsement of someone's public key
  - A certificate contains at least two things: The **identity** of the person, and the **key**
- Abbreviated notation
  - Encryption under a public key *PK*: {"Message"}$_{PK}$
  - Signing with a private key *SK*: {"Message"}$_{SK^{-1}}$
    - Recall: A signed message must contain the message along with the signature; you can't send the signature by itself!
- Scenario: Alice wants Bob's public key. Alice trusts EvanBot (PK_E, SK_E)
  - EvanBot is our trust anchor
  - If we trust PK_E, a certificate we would trust is {"Bob's public key is PK_B"}$_{SK_E^{-1}}$

Attempt #1: The Trusted Directory

- Idea: Make a central, trusted directory (TD) from where you can fetch anybody's public key
  - The TD has a public/private keypair PK_{TD}, SK_{TD}
  - The directory publishes PK_{TD} so that everyone knows it (baked into computers, phones, OS, etc.)
  - When you request Bob's public key, the directory sends a certificate for Bob's public key
    - {"Bob's public key is PK_B"}$_{SK_{TD}^{-1}}$
  - If you trust the directory, then now you trust every public key from the directory
- What do we have to trust?
  - We have received TD's key correctly
  - TD won't sign a key without verifying the identity of the owner

Attempt #1: The Trusted Directory

- Let's say that Michael Drake (MD, President of UC) runs the TD
  - We want Nick Weaver's public key: Ask MD
  - We want David Wagner's public key: Ask MD
  - We want Raluca Ada Popa's public key: Ask MD
  - MD has better things to do (like making sure his private key isn't stolen)!
- Problems: Scalability
  - One directory won't have enough compute power to serve the entire world
- Problem: Single point of failure
  - If the directory fails, *cryptography stops working*
  - If the directory is compromised, you can't trust anyone
  - If the directory is compromised, it is difficult to recover

Certificate Authorities

- Addressing scalability: Hierarchical trust
  - The roots of trust may **delegate** trust and signing power to other authorities
    - {“Carol Christ’s public key is PK_{CC}, and I trust her to sign for UCB”}$_{SK_{MD}^{-1}}$
    - {“Dave Wagner’s public key is PK_{DW}, and I trust him to sign for the CS department”}$_{SK_{CC}^{-1}}$
    - {“Nick Weaver’s public key is PK_{NW} (but I don’t trust him to sign for anyone else)”}$_{SK_{DW}^{-1}}$
  - MD is still the root of trust (**root certificate authority**, or **root CA**)
  - CC and DW receive delegated trust (**intermediate CAs**)
  - NW’s identity can be trusted
- Addressing scalability: Multiple trust anchors
  - There are ~150 root CAs who are implicitly trusted by most devices
  - Public keys are hard-coded into operating systems and devices
  - Each delegation step can restrict the scope of a certificate’s validity
  - Creating the certificates is an *offline* task: The certificate is created once in advance, and then served to users when requested

Revocation

- What happens if a certificate authority messes up and issues a bad certificate?
  - Example: {“Bob’s public key is PK_M”}$_{SK_{CA}^{-1}}$
  - Example: Verisign (a certificate authority) accidentally issued a certificate saying that an average Internet user’s public key belonged to Microsoft

Revocation: Expiration Dates

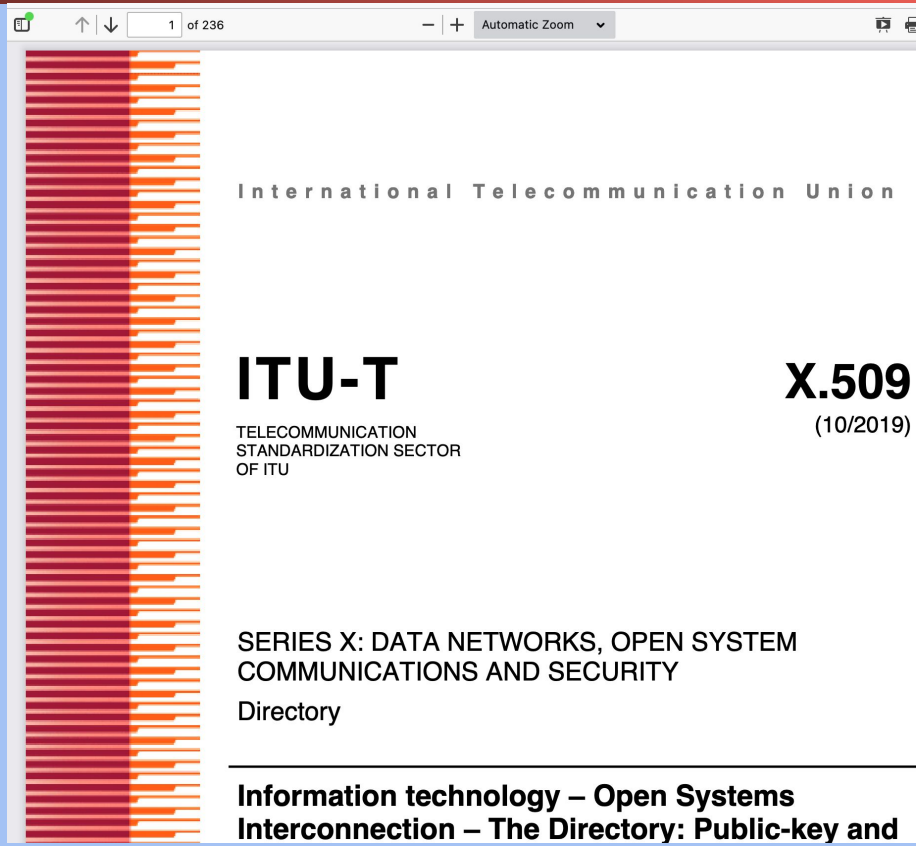
- Approach #1: Each certificate has an expiration date
  - When the certificate expires, request a new certificate from the certificate authority
  - The bad certificate will eventually become invalid once it expires
- Benefits
  - Mitigates damage: Eventually, the bad certificate will become harmless
- Drawbacks
  - Adds management burden: Everybody has to renew their certificates frequently
  - If someone forgets to renew a certificate, their website might stop working
- Tradeoff: How often should certificates be renewed?
  - Frequent renewal: More secure, less usable
  - Infrequent renewal: Less secure, more usable
- LetsEncrypt (a certificate authority) chose very frequent renewal
  - It turns out frequent renewal is more usable:
    It forces automated renewal instead of a once-every 3 year task that gets forgotten!

Revocation: Announcing Revoked Certificates

- Approach #2: Periodically release a list of invalidated certificates
  - Users must periodically download a Certification Revocation List (CRL)
- How do we authenticate the list?
  - The certificate authority signs the list!
    - {“The certificate with serial number 0xdeadbeef is now revoked”}$_{SK_{CA}^{-1}}$
- Drawbacks
  - Lists can get large
    - Mitigated by shorter expiration dates (don’t have to list them once they expire)
  - Until a user downloads a list, they won’t know which certificates are revoked
- What happens if the certificate authority is unavailable?
  - Fail-safe default: Assume all certificates are invalid? Now we can’t trust anybody!
    - Possible attack: Attacker forces the CA to be unavailable (denial of service attack)
  - Use old list: Potentially dangerous if the old list is missing newly revoked certificates

Certificates: Complexity

- Certificate protocols can get very complicated
  - Example: X.509 is incredibly complicated (a 236 page standard!) because it tried to do everything



Alternative: Web of Trust

- Modern public-key infrastructures are structured like trees
- Originally, public-key infrastructures looked like graphs instead
  - Everybody can issue certificates for anyone else
  - Example: Alice signs Bob's key. Bob signs Carol's key. If Dave trusts Alice, he trusts Bob and Carol.
  - Benefit: You know the trust anchor personally (e.g. because you met them in-person, or because you signed their key)
  - Problem: Graphs get far more complex than trees!
- OpenPGP (Pretty Good Privacy) originally used the web of trust model
  - Key-signing parties: meeting in-person to sign each other's public keys
  - It quickly proved to be a disaster
  - Instead, everyone just relies on MIT's central keyserver which is broken!
- **Takeaway**: Trust anchors make public-key infrastructures much simpler!

Summary: Certificates

- Certificates: A signed attestation of identity
- Trusted directory: One server holds all the keys, and everyone has the TD's public key
  - Not scalable: Doesn't work for billions of keys
  - Single point of failure: If the TD is hacked or is down, cryptography is broken
- Certificate authorities: Delegated trust from a pool of multiple root CAs
  - Root CAs can sign certificates for intermediate CAs
  - Revocation: Certificates contain an expiration date
  - Revocation: CAs sign a list of revoked certificates

Password Hashing

Textbook Chapter 14

Review: Cryptographic Hashes

- Hashes accept arbitrarily large inputs
- Hashes “look” random
  - Change a single bit on the input and each output bit has a 50% chance of flipping
  - And until you change the input, you can't predict which output bits are going to change
- The ones we talked about are *fast*
  - Can operate at many many MB/s: Faster at processing data than block ciphers
- Recall: Security properties
  - One way: Given an output y, it is infeasible to find any input x such that $H(x) = y$.
  - Collision resistant: It is infeasible to find another any pair of inputs $x' \neq x$ such that $H(x) = H(x')$.

Storing Passwords

- Password: A secret string a user types in to prove their identity
  - When you create an account with a service: Create a password
  - When you later want to log in to the service: Type in the same password again
- How does the service check that your password is correct?
- Bad idea #1: Store a file listing every user's password
  - Problem: What if an attacker hacks into the service? Now the attacker knows everyone's passwords!
- Bad idea #2: Encrypt every user's password before storing it
  - Problem: The attacker could steal the passwords file *and* the key and decrypt everyone's passwords!
- We need a way to verify passwords *without* storing any information that would allow someone to recover the original password

Password Hashing

- For each user, store a *hash* of their password
- Verification process
  - Hash the password submitted by the user
  - Check if it matches the password hash in the file
- What properties do we need in the hash?
  - Deterministic: To verify a password, it has to hash to the same value every time
  - One-way: We don’t want the attacker to reverse hashes into original passwords

Password Hashing: Attacks

- What if two different users decide to use **`password123`** as their password?
  - Hashes are deterministic: They'll have the same password hash
  - An attacker can see which users are using the same password
- Brute-force attacks
  - Most people use insecure, common passwords
  - An attacker can pre-compute hashes for common passwords: *H*(**`"password123"`**), *H*(**`"password1234"`**), *H*(**`"1234567890"`**), etc.
  - **Dictionary attack**: Hash an entire dictionary of common passwords
- **Rainbow tables**: An algorithm for computing hashes that makes brute-force attacks easier

Salted Hashes

- Solution #1: Add a unique, random salt for each user
- **Salt**: A random, public value designed to make brute-force attacks harder
  - For each user, store: username, salt, *H*(password || salt)
  - To verify a user: look up their salt in the passwords file, compute *H*(password || salt), and check it matches the hash in the file
  - Salts should be long and random
  - Salts are not secret (think of them like nonces or IVs)
- Brute-force attacks are now harder
  - Assume there are *M* possible passwords and *N* users in the database
  - Unsalted database: Hash all possible passwords, then lookup all users' hashes ⇒ $O(M + N)$
  - Salted database: Hash all passwords for each user's salt ⇒ $O(MN)$

Slow Hashes

- Solution #2: Use slower hashes
- Cryptographic hashes are usually designed to be fast
  - SHA is designed to produce a checksum of your 1 GB document as fast as possible
- Password hashes are usually designed to be slow
  - Legitimate users only need to submit a few password tries. Users won't notice if it takes 0.0001 seconds or 0.1 seconds for the server to check a password.
  - Attackers need to compute millions of hashes. Using a slow hash can slow the attacker by a factor of 1,000 or more!
  - Note: We are not changing the asymptotic difficulty of attacks. We're adding a large constant factor, which can have a huge practical impact for the attacker

Slow Hashes: PBKDF2

- **Password-based key derivation function 2 (PBKDF2)**: A slow hash function
  - Setting: An underlying function that outputs random-looking bits (e.g. HMAC-SHA256)
  - Setting: The desired length of the output (*n*)
  - Setting: Iteration count (higher = hash is slower, lower = hash is faster)
  - Input: A password
  - Input: A salt
  - Output: A long, random-looking *n*-bit string derived from the password and salt
  - Implementation: Basically computing HMAC 10,000 times
- Benefits (assuming the user password is strong)
  - Derives an arbitrarily long string from the user's password
  - Output can be directly used as a symmetric key
  - Output can also be used to seed a PRNG or generate a public/private key pair
  - Algorithm is slow, but doesn't use a lot of memory (alternatives like Scrypt and Argon2 use more memory)

Offline and Online Attacks

- **Offline attack**: The attacker performs all the computation themselves
  - Example: Mallory steals the password file, and then computes hashes herself to check for matches.
  - The attacker can try a huge number of passwords (e.g. use many GPUs in parallel)
  - Defenses: Salt passwords, use slow hashes
  - If an attacker can do an offline attack, you need a really strong password (e.g. 7 or more random words)
- **Online attack**: The attacker interacts with the service
  - Example: Mallory tries to log in to a website by trying every different password. Mallory is forcing the server to compute the hashes.
  - The attacker can usually only try a few times per second, with no parallelism
  - Defenses: Add a timeout or rate limit the number of tries to prevent the attacker from trying too many times

Summary: Password Hashing

- Store hashes of passwords so that you can verify a user's identity without storing their password
- Attackers can use brute-force attacks to learn passwords (especially when users use weak passwords)
  - Defense: Add a different **salt** for each user: A random, public value designed to make brute-force attacks harder
- **Offline attack**: The attacker performs all the computation themselves
  - Defense: Use salted, slow hashes instead of unsalted, fast hashes
- **Online attack**: The attacker interacts with the service
  - Defense: Use timeouts

Traffic Analysis & Side Channels

Traffic Analysis & Side Channels

- **Traffic analysis**: Analyzing who is talking to whom and when
  - The encryption schemes we'll be studying do not hide the identity of who you're talking to
  - The information used for this analysis is often referred to as **metadata**: Data *about* the message and its context
- **Side channels**: Information about the plaintext revealed as a result of the *implementation* of the scheme, not the scheme itself
  - Modern crypto systems are usually broken through side channels

Traffic Analysis & Side Channels in Practice: Spies

- In the 1990s, there were some Russian spies in the US
  - The TV series “The Americans” was based on this incident
- A Cuban number station had a bug: some nights it never broadcasted “9”
  - Normally, 0–9 would be equally frequent
- It turns out this corresponded to when the Russian spies were on vacation
  - The way that random numbers were generated for cover traffic had a bug in it
  - The FBI used this as part of their investigation
- **Takeaway**: Secure algorithms can be broken in insecure implementations, leaking information

Nothing-Up-My-Sleeve-Numbers

Nothing-Up-My-Sleeve-Numbers

- Cryptography uses a lot of constants
  - Initial state for SHA
  - Prime *p* and generator *g* in Diffie-Hellman
  - Parameters for elliptic-curve cryptography (curve equations, points on the curve like *P* and *Q*)
  - *ipad* and *opad* in HMAC
- Usually, any value could work, but the designer needed to choose *some* constant for the algorithm
  - Example: In Diffie-Hellman, any large prime *p* and generator *g* works, but there's a default *p* and *g* that everyone uses
- Where do these default parameter values come from?

PRNG Sabotage: Dual_EC_DRBG

- Dual_EC_DRBG: A PRNG published by the NSA behind the scenes
- Relies on two public hard-coded parameters P and Q
  - P and Q are points on an elliptic curve
  - If P and Q are related by $Q = eP$ (analogous to $Q = P^e \bmod n$ in discrete log), you can learn the internal state!
- It also sucked!
  - It was horribly slow (much slower than HMAC or CTR PRNGs)
  - It had subtle biases that shouldn't exist in a secure PRNG: You could distinguish the upper bits from random
  - Cryptographers spotted these flaws early on
  - Why would anyone use such a horrible PRNG?

PRNG Sabotage: Dual_EC_DRBG

- Why would anyone use such a horrible PRNG?
- Story time:
  - RSA Data Security accepts $10 million from the NSA
  - In exchange, RSA Data Security implements Dual_EC in their RSA BSAFE library, and silently makes it the default PRNG
  - With RSA Data Security's support, Dual_EC became a NIST standard
    - Used in other products
  - 2013: Whistleblower Edward Snowden reveals classified NSA information
    - The New York Times vaguely mentions a crypto talk given by Microsoft people
    - Everybody quickly realized this referred to a backdoor the NSA inserted into Dual_EC
    - Backdoor: An attack inserted by an organization (e.g. NSA) so that they, but nobody else, can attack the system

PRNG Sabotage: Dual_EC_DRBG

- With the backdoor, it's possible to predict both future and past PRNG outputs
  - No rollback resistance, unlike HMAC-DRBG
- Recall the attack when a PRNG is not rollback resistant:
  - Generate a secret key
  - Generate some other publicly visible random value (e.g. IVs, nonces, an NSA-sponsored "standard" that requires some random output)
  - Since the PRNG is not rollback-resistant, the NSA can view the public random value and use the backdoor to learn secret keys

PRNG Sabotage: Dual_EC_DRBG

- Juniper Networks used Dual_EC in their virtual private networks (VPNs)
- Juniper claims their version of Dual_EC was safe from the NSA backdoor
  - Juniper selected a different public parameter (*Q*) than the NSA's public parameter
- Later: Juniper is hacked
  - The hacker changed Dual_EC's public parameter (*Q*) to give themselves a backdoor!
    - And now the original backdoor owner can't get in…
- Later: Juniper is hacked again
  - The hacker adds an SSH backdoor
- Later: Juniper notices the SSH backdoor
  - … and also notices the changed Dual_EC backdoor
  - So they release an update to patch the Dual_EC backdoor
- Later: Everyone receives an update
  - Everyone: "Ohh, patch for a backdoor. Let's see what got fixed. Oh, these look like Dual_EC parameters…"

Diffie-Hellman Sabotage?

- Another case of potential sabotage
- 1024b Diffie-Hellman is moderately impractical
- However, for a specially chosen *p*, cracking Diffie-Hellman becomes 1 million times easier
  - Recall: *p* is a public value. Most implementations use default hard-coded values of *p*
  - The most commonly-used “example” *p* has unknown origin! (lost to time)

Something Up Their Sleeves… (Maybe)

- Mysterious public parameters can cast doubt *even when a design is solid*
- Recall DES (the block cipher used before AES)
  - The DES standard was developed by IBM but with input from the NSA
  - Everyone was suspicious that the NSA tampered with S-boxes (hard-coded constant values in the block cipher algorithm)
  - It turns out they did: The NSA made them stronger against an attack that they knew about but the public didn't know about
- P-256 and P-384: Two elliptic curves defined by the NSA
  - Remember: Elliptic curves are public parameters in elliptic-curve cryptography
  - The elliptic curves were chosen mysteriously and cast doubt on whether there is a backdoor

Takeaway: Nothing-Up-My-Sleeve-Numbers

- Good systems should clearly and transparently describe how public parameters are generated
  - Argue about why these values provide good security (e.g. AES designers argued this for the values in their S-boxes)
  - Choose values with obvious human significance: The developer probably doesn't have millions of values of obvious significance to brute-force and pick a value with a backdoor
    - Seed a PRNG with your name
    - The first few digits of π
    - 0x67452301, 0xefcdab89, … (SHA-1)
    - Fractional parts of the square/cube roots of prime numbers (SHA-2)

Case Study: iPhone Security

iPhone Security

- Apple's security philosophy:
  - In your hands, you can access everything on your phone
  - In anybody else's hands, the phone is an inert "brick" (nothing can be accessed)
- Apple uses a small co-processor in the phone to handle all cryptography
  - The "Secure Enclave" (recall: small TCB)
- The rest of the phone is untrusted
  - Memory is untrusted, so all data must be encrypted
  - The CPU must ask the Secure Enclave to decrypt data
  - Some data (e.g. credit card information for Apple Pay) is only readable by the Secure Enclave
- Effaceable Storage
  - Data is often stored in multiple places for redundancy, or not entirely wiped on deletion (for speed)
  - Effaceable storage: A section of memory where if memory is wiped, it is guaranteed to be gone
  - Requires some electrical engineering trickery to implement

iPhone Security

- A lot of keys encrypted by keys
  - There is a random master key, K_{phone}, that can be used to decrypt all the other keys
- K_{phone} is encrypted by the user's password and stored in flash memory
  - Run PBKDF on the password to get K_{user}, and use K_{user} to encrypt K_{phone}
- How do we prevent an offline brute-force attack?
  - Include a random 256-bit secret hardcoded into the Secure Enclave for encryption
  - The only way to get this secret is to take apart the chip!
  - The Secure Enclave can’t read the secret, but can only use it as an input for hardware cryptography
  - The user key K_{user} is actually function of the password and Enc(secret, password)
  - Offline attacks are not possible without the secret
- **Takeaway**: Mixing in on-chip secret requires online attacks!
  - If the attacker doesn't know the on-chip secret, they can’t perform an offline attack

iPhone Security

- How do we prevent online brute-force attacks?
  - Online attacks must go through the secure enclave
  - Timeouts: After 5 tries, add a delay to slow down each try
  - After 10 tries, optionally wipe K_{phone} forever
  - Remember: K_{phone} is the root key for decrypting everything else. Erasing K_{phone} effectively erases everything on the phone!
  - Even if an attacker compromises the secure enclave, guessing is still limited to 10 tries per second
- **Takeaway**: Timeouts and slow algorithms prevent online attacks. If possible, limit attackers to online attacks

iPhone Security: Backups

- A necessary weakness: Backups
  - Backup: Copying all data from the phone somewhere else
  - The data is copied unencrypted (decrypted with the secret on the chip)
  - Backups are necessary so you can recover your data on another phone
- Attack: Use backups to steal your data
  - The attacker finds a way to unlock your phone without your password (e.g. hold it to your face or use your fingerprint)
  - Remember: consider human factors!
  - The attacker syncs your phone with a new computer
- Change of policy as of iOS 11
  - To create a backup on a new computer, you need to enter your password to trust the computer
  - Previous attack is no longer possible: can't create a backup without knowing the password

Case Study: Samsung

- Samsung has a similar concept
  - But their implementation has, umm, issues…
- Guess with, GCM with IV reuse!
  - And although “fixed”, you could do a downgrade attack to cause the fixed version to still reuse IVs
- **Takeaway**: CTR mode is dangerous when used improperly…and even experts misuse it!

Cryptology ePrint Archive: Report 2022/208 Link

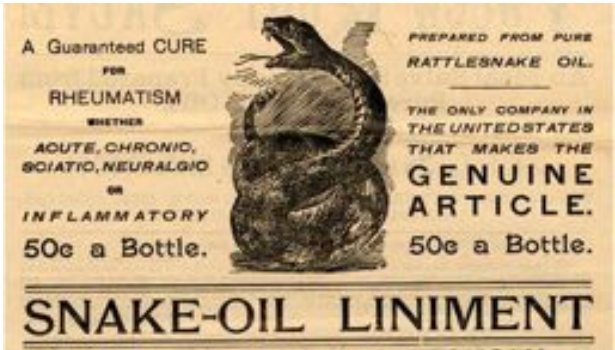
Alon Shakevsky and Eyal Ronen and Avishai Wool *February 20, 2022*

In this work, we expose the cryptographic design and implementation of Android's Hardware-Backed Keystore in Samsung's Galaxy S8, S9, S10, S20, and S21 flagship devices. We reversed-engineered and provide a detailed description of the cryptographic design and code structure, and we unveil severe design flaws. We present an IV reuse attack on AES-GCM that allows an attacker to extract hardware-protected key material, and a downgrade attack that makes even the latest Samsung devices vulnerable to the IV reuse attack. We demonstrate working key extraction attacks on the latest devices. We also show the implications of our attacks on two higher-level cryptographic protocols between the TrustZone and a remote server: we demonstrate a working FIDO2 WebAuthn login bypass and a compromise of Google’s Secure Key Import.

Case Study: Snake Oil Cryptography

Snake Oil

- Original meaning: Fraudulent “cure-all” medicines sold in the 1700s and 1800s
  - Sellers promised buyers that snake oil cures all diseases (even though it doesn’t)
  - Took advantage of uninformed buyers and lack of regulations
- Modern meaning: Scams
  - Using deceptive advertising to trick uninformed buyers into buying useless products
- **Snake oil security** (**snake oil cryptography**): Useless security products advertised to uninformed buyers



Signs of Snake Oil Cryptography

- Amazingly long key lengths
  - Once brute-forcing a key becomes astronomically hard, making it longer probably doesn't provide extra security
  - The NSA is super paranoid, and even they don’t use >256-bit symmetric keys or >4,096-bit public keys
- New algorithms and wild protocols
  - There is no reason to use a brand-new block cipher, hash algorithm, public-key algorithm, etc.
  - Existing protocols have been vetted by security experts for years: They’re widespread for a good reason!
  - New protocols probably means someone is trying to write their own crypto (and asking for trouble!)
- Fancy-sounding technical buzzwords
  - Claims of inventing “new math”

Signs of Snake Oil Cryptography

- “One time pads”
  - Recall: One-time pads are secure if you never reuse the key
  - Recall: Secure one-time pads are highly impractical
  - Almost all schemes advertised as “one-time pads” probably aren't true one-time pads
  - Wacky stream ciphers (often self-designed) are often advertised as “one-time pads”
- Rigged “cracking contests”
  - Advertising a secure scheme by challenging the public to break the scheme
  - The challenge is often “decrypt this message” with no context or structure
  - Example: Telegram offered a $300,000 prize in a contest to break their encryption

Snake Oil Example: Crown-Sterling

ars TECHNICA

Link

Alleged “snake oil” crypto company sues over boos at Black Hat

Sean Gallagher *August 23, 2019*

Crown Sterling seeks damages after attendee disrupts "controversial" talk on prime prediction.

Crown Sterling: A snake-oil cryptography company
Black Hat: A security research conference

Snake Oil Example: Crown-Sterling

ars TECHNICA

Link

Alleged “snake oil” crypto company sues over boos at Black Hat

Sean Gallagher *August 23, 2019*

Grant's presentation, entitled "Discovery of Quasi-Prime Numbers: What Does this Mean for Encryption," was based on a paper called "Accurate and Infinite Prime Prediction from a Novel Quasi-PrimeAnalytical Methodology." That work was published in March of 2019 through Cornell University's arXiv.org by Grant's co-author Talal Ghannam—a physicist who has self-published a book called The Mystery of Numbers: Revealed through their Digital Root as well as a comic book called The Chronicles of Maroof the Knight: The Byzantine. The paper, a slim five pages, focuses on the use of digital root analysis (a type of calculation that has been used in occult numerology) to rapidly identify prime numbers and a sort of multiplication table for factoring primes.

Takeaway: Buzzwords are signs of snake oil cryptography

Snake Oil Example: Crown-Sterling

ars TECHNICA

Link

Snake oil or genius? Crown Sterling tells its side of Black Hat controversy

Sean Gallagher *August 29, 2019*

How does Time AI work?

So how is Time AI said to work? Crown Sterling's website describes Time AI as "a dynamic non-factor based quantum encryption utilizing multidimensional encryption technology including time, music's infinite variability, artificial intelligence, and most notably mathematical constants to generate entangled key pairs."

Takeaway: Brand-new math and buzzwords are signs of snake oil cryptography

Snake Oil Example: Crown-Sterling

ars TECHNICA

Link

Medicine show: Crown Sterling demos 256-bit RSA key-cracking at private event

Sean Gallagher *September 20, 2019*

Demo of crypto-cracking algorithm fails to convince experts.

Takeaway: Flashy demonstrations are signs of snake oil cryptography

Snake Oil Example: Crown-Sterling

ars TECHNICA

Link

Medicine show: Crown Sterling demos 256-bit RSA key-cracking at private event

Sean Gallagher *September 20, 2019*

Nicholas Weaver, lecturer at the University of California Berkeley's Department of Electrical Engineering and Computer Sciences, reacted to Grant's latest demonstration with this statement to Ars:

> It was previously an open question whether Mr Grant was a fraud or just delusional. His new press release now makes me certain he is a deliberate fraud.
>
> He received a lot of feedback from cryptographers, both polite and rude, so showing this level of continued ignorance is willful at this point. His video starts with the ridiculously false notion that factoring is all there is for public key. He then insists that breaking a 256 bit RSA key or even a 512b key is somehow revolutionary. It's not. Professor [Nadia] Heninger at UCSD, as part of her work on the FREAK attack, showed that factoring a 512 bit key is easily accomplished with less than $100 of computing time in 2015.
>
> His further suggesting that breaking 512-bit breaks RSA is also ridiculous on its face. Modern RSA is usually 2048 bits or higher, and there is a near-exponential increase in the difficulty of factoring with the number of bits.
>
> At this point I have to conclude he is an outright fraud, and the most likely explanation is he's looking to raise investment from ignorant accredited investors. And now I wonder how many other companies he's started are effectively fraudulent.

Snake Oil Example: Crown-Sterling

Nicholas Weaver ✓
@ncweaver

Link

September 21, 2019

FYI My offer stands you litigious fraudulent fuckwits. If you consider my statements that you are fraudulent fuckwits based on this release & demonstration libel, I'll gladly tell you a good address for service, just DM for info.

Example: Cryptocurrency Snake Oil

- IOTA: A cryptocurrency designed for the Internet of Things (IoT)
  - Uses a hash-based scheme instead of standard public key signatures, meaning you can never reuse a key
  - 10,000-bit signatures (compare with 450-bit RSA signatures, which are considered big)
  - Created their own hash function… that was quickly broken
  - Claims to be a distributed system, but relies entirely on a central authority (not distributed)
  - Uses trinary math? (Requiring entirely new processors?)
- **Takeaway**: Be able to recognize snake oil cryptography