Practical Crypto, Random Numbers, Crypto Fails


Cryptography is nightmare magic math that cares what kind of pen you use -@swiftonsecurity

Announcements!

- Midterm 1 Monday, 5-7 pm
  - Bring your student ID
- Project 1 due tomorrow
  - Make only 1 submission per group!

In Practice: Session Keys...

- You use the public key algorithm to encrypt/agree on a session key..
  - And then encrypt the real message with the session key
  - You ***never*** actually encrypt the message itself with the public key algorithm
- Why?

How to prevent a MitM attack?

- Digital signatures?
- If Bob knows Alice's key, and Alice knows Bob's...
  - How will be "next time"
- Alice doesn't just send a message to Bob...
  - But creates a random key **k**...
  - Sends $\mathbf{E(M, K_{sess})}$, $\mathbf{E(K_{sess}, B_{pub})}$, $\mathbf{S(H(M), A_{priv})}$
- Only Bob can decrypt the message, and Bob can verify the message came from Alice
  - So Mallory is SOL!

Signatures Enable Ephemeral Diffie/Hellman

- Bob knows (somehow) Alice's public key...
  - We will find out how later when we talk about ***certificates***
  - Or, as in the project, the "trusted keystore" can tell you Alice's public key
- Now Alice doesn't just send $\mathbf{g^a}$, but also $\mathbf{sign(g^a,K_{alice})}$
  - As a consequence, now Mallory can't play the MitM!
- And yet we have "forward secrecy"
  - Even if Eve gets Alice's private key, she can't decrypt old messages or new messages
  - Even if Malory gets Alice's private key, he can only intercept new messages as a man-in-the-middle

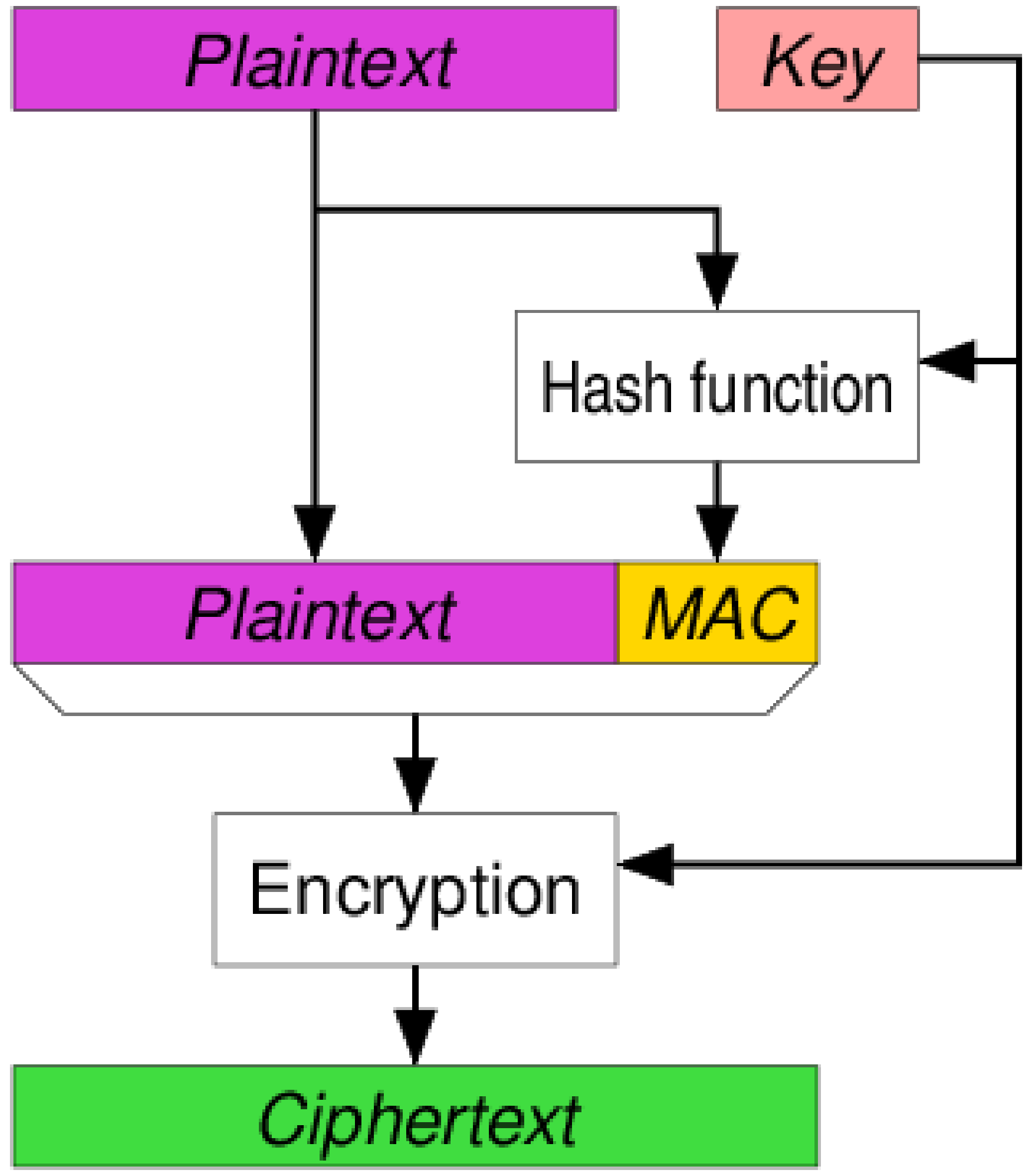
Exercise:
Send me an encrypted message

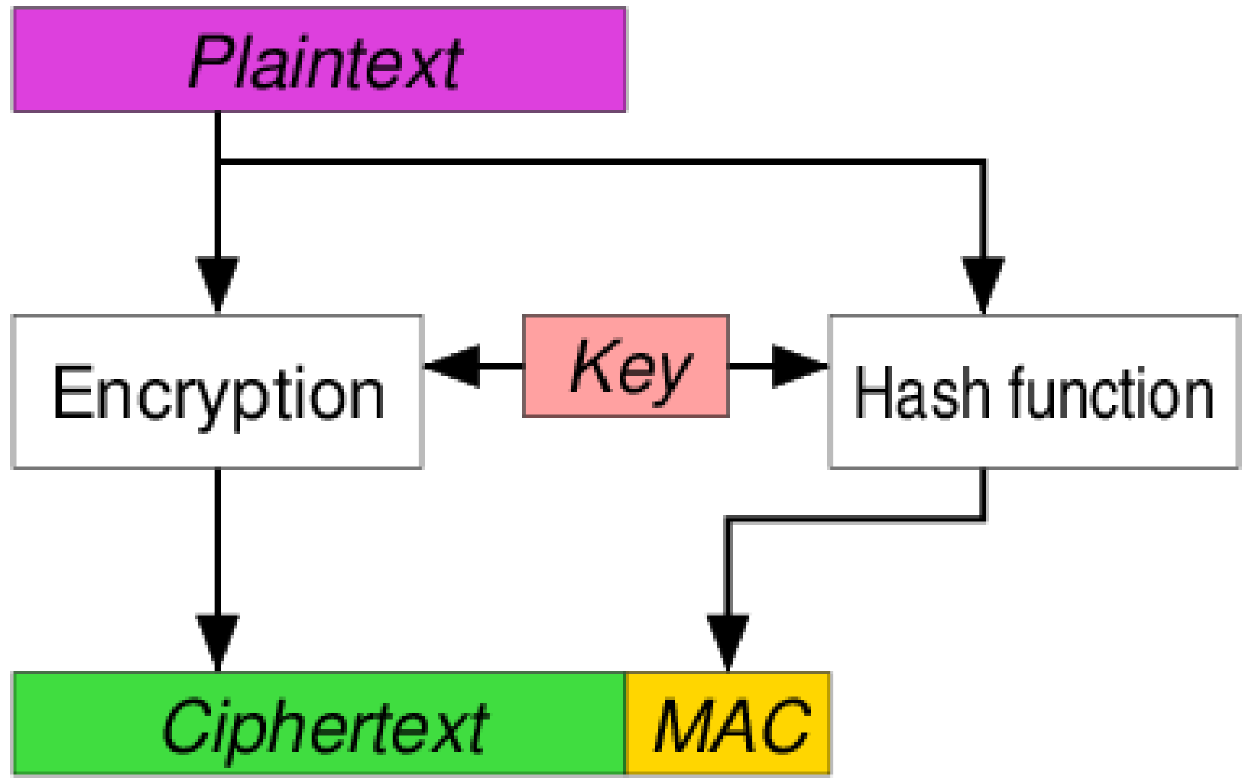
- Make sure no one else can read the message
  - Use any communication method you want
- How can you find my public key?
  - How can you be sure it's me?
  - How can I be sure it's you?
- How can I respond in encrypted form?
- Does the communication have forward secrecy?
- Does it have integrity? Authentication?
- Is it deniable? Or non-repudiable?

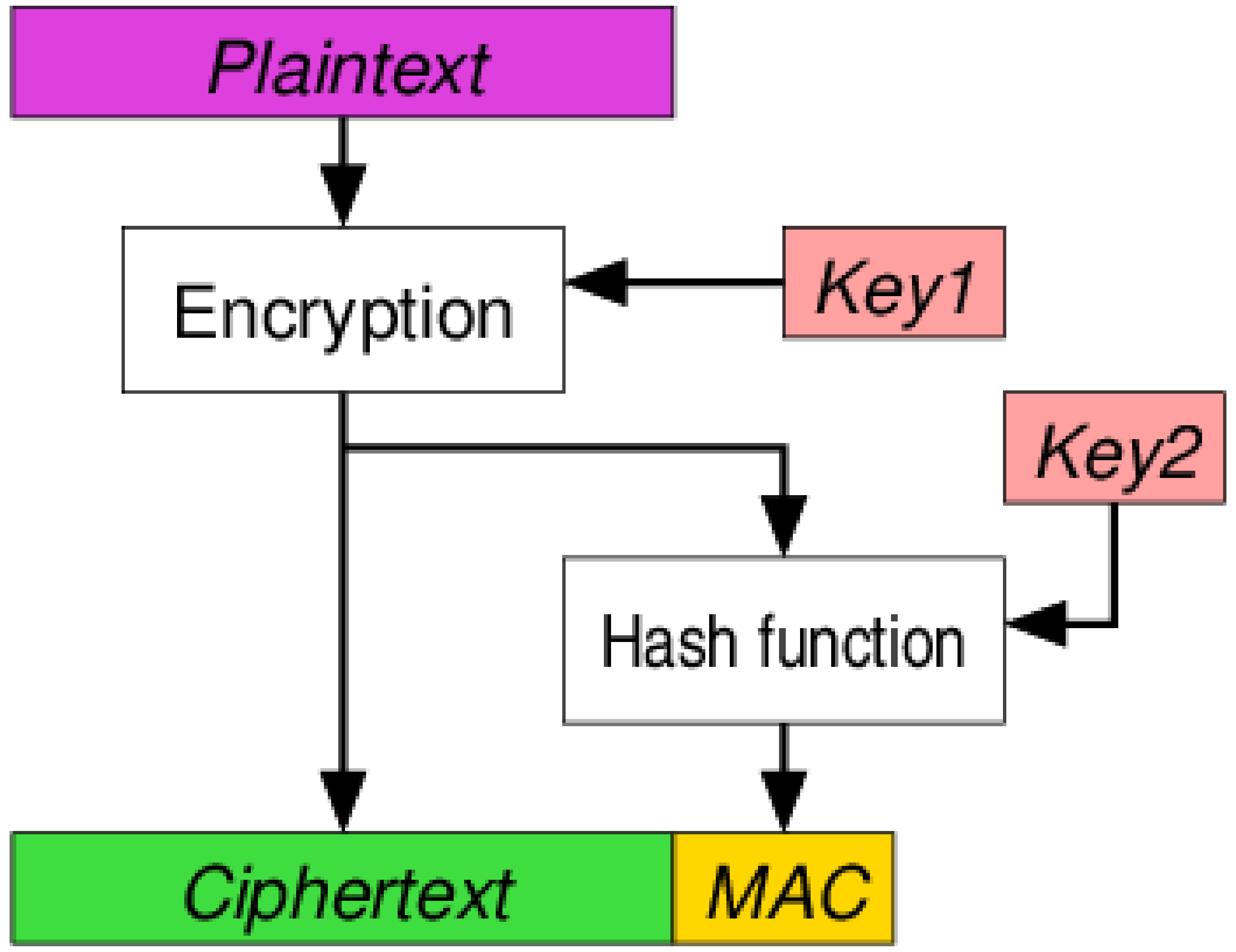
Cryptofail: MAC then Encrypt or Encrypt then MAC?

- You should ***never*** use the same key for the MAC and the Encryption
  - Some MACs will break completely if you reuse the key
  - Even if it is ***probably*** safe (eg, AES for encryption, HMAC for MAC) its still a bad idea
- MAC then Encrypt:
  - Compute **T = MAC(M,K_{mac})**, send **C = E(M||T,$K_{encrypt}$)**
- Encrypt and MAC:
  - Compute **C = E(M,$K_{encrypt}$)**, **T = MAC(M,K_{mac})**, send **C||T**
- Encrypt then MAC
  - Compute **C = E(M,$K_{encrypt}$)**, **T = MAC(C,K_{mac})**, send **C||T**

Cryptofail: MAC then Encrypt or Encrypt then MAC?



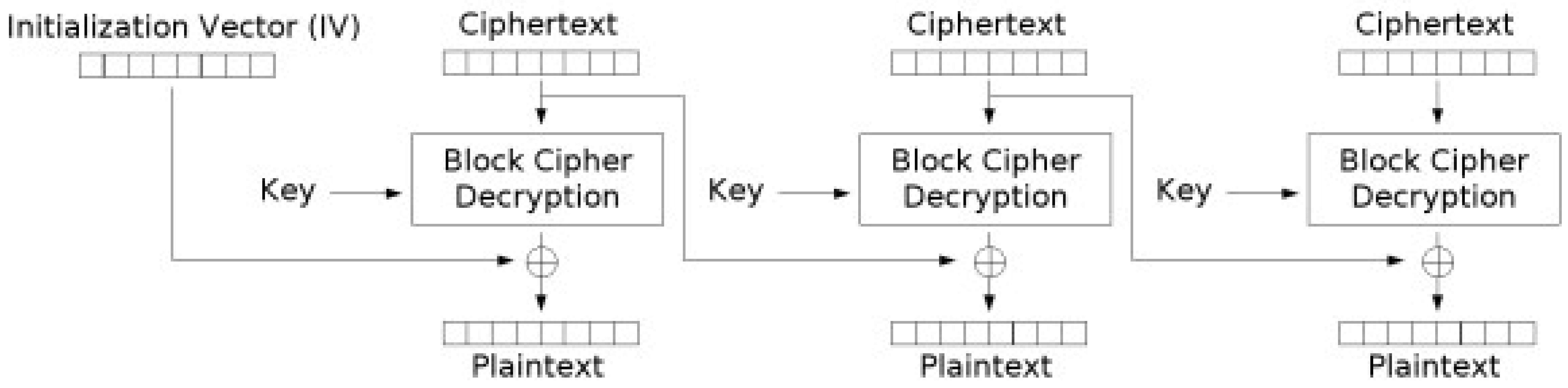




- MAC then Encrypt
- Encrypt and MAC
- Encrypt then MAC

Padding Oracle Attack

- Can deterministically modify last padding byte



Cipher Block Chaining (CBC) mode decryption

The TLS 1.0 "Lucky13" Attack: "F-U, This is Cryptography"

- HTTPS/TLS uses MAC then Encrypt
  - With CBC encryption
- The Lucky13 attack changes the cipher text in an attempt to discover the state of a byte
  - But can't predict the MAC
  - The TLS connection retries after each failure so the attacker can try multiple times
    - Goal is to determine the status each byte in the authentication cookie which is in a known position
- It detects the ***timing*** of the error response
  - Which is different if the guess is right or wrong
    - Even though the underlying algorithm was "***proved***" secure!
- So always do Encrypt then MAC since, once again, it is more mistake tolerant

CryptoFail: Side Channels

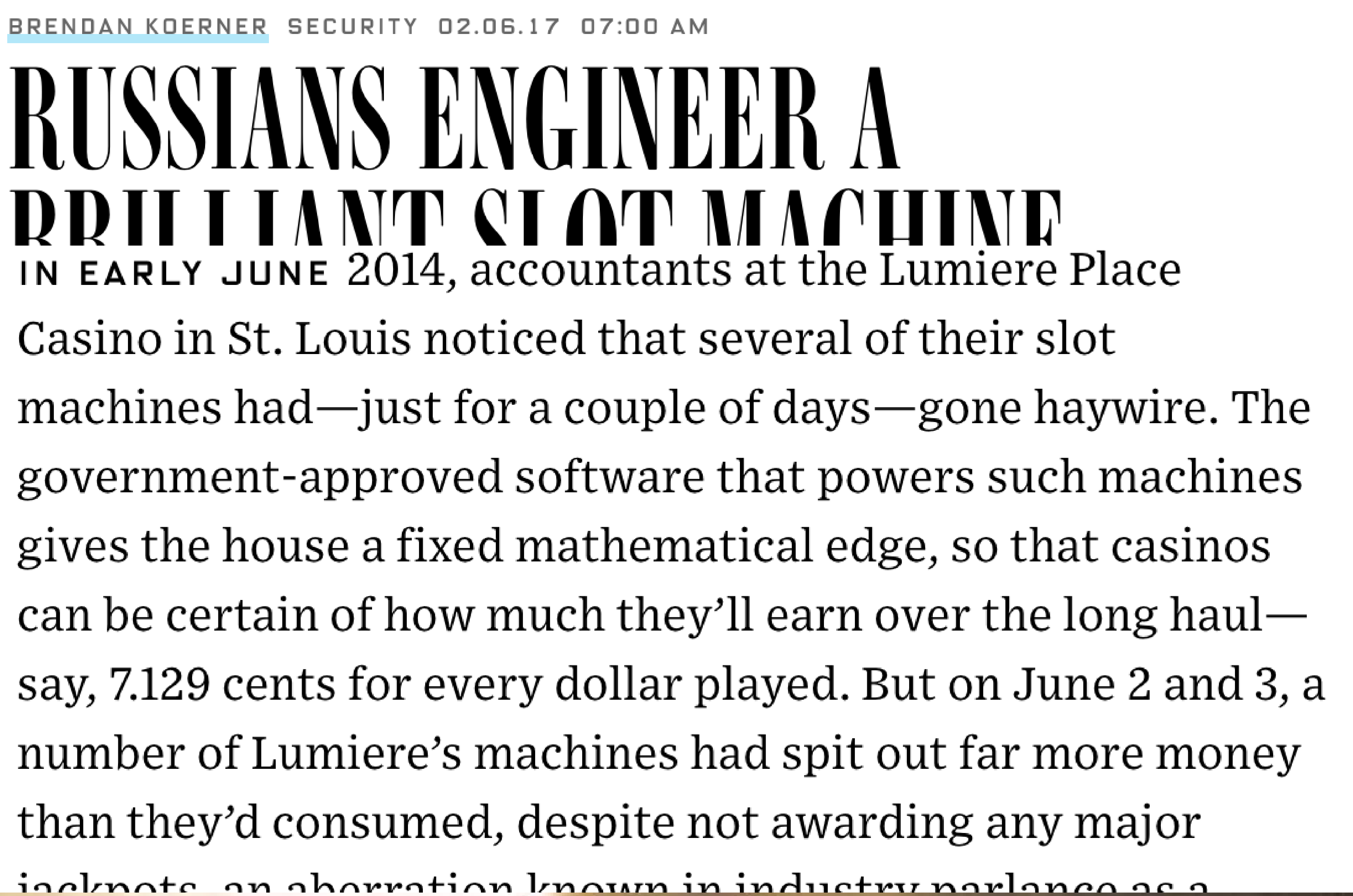
- Anything outside the normal message
  - The ***time*** it takes to decrypt a message (or even just report an error)
  - The ***power*** it takes to decrypt a message
  - The ***cache state*** of a processor after another process completes encryption
  - Electromagnetic radiation when encrypting
    - TEMPEST attacks
- These are often how you break crypto systems in practice

A Lot of Uses for Random Numbers...

- The key foundation for all modern cryptographic systems is often not encryption but these "random" numbers!
- So many times you need to get something random:
  - A random cryptographic key
  - A random initialization vector
  - A random "nonce" (use-once item)
  - A unique identifier
  - Stream Ciphers
- If an attacker can ***predict*** a random number things can catastrophically fail

Breaking Slot Machines

- Some casinos experienced unusual bad "luck"
  - The suspicious players would wait and then all of a sudden try to play
- The slot machines have ***predictable*** pRNG
  - Which was based on the current time & a seed
- So play a little...
  - With a cellphone watching
  - And now you know when to press "spin" to be more likely to win
- Oh, and this ***never*** affected Vegas!
  - ***Evaluation standards*** for Nevada slot machines specifically designed to address this sort of issue



BRENDAN KOERNER SECURITY 02.06.17 07:00 AM

RUSSIANS ENGINEER A

IN EARLY JUNE 2014, accountants at the Lumiere Place Casino in St. Louis noticed that several of their slot machines had—just for a couple of days—gone haywire. The government-approved software that powers such machines gives the house a fixed mathematical edge, so that casinos can be certain of how much they'll earn over the long haul—say, 7.129 cents for every dollar played. But on June 2 and 3, a number of Lumiere's machines had spit out far more money than they'd consumed, despite not awarding any major

Breaking Bitcoin Wallets

- blockchain.info supports "web wallets"
  - Javascript that protects your Bitcoin
- The private key for Bitcoin needs to be random
  - Because otherwise an attacker can spend the money
- An "Improvment" [sic] to the RNG reduced the entropy (the actual randomness)
  - Any wallet created with this improvment was brute-forceable and could be stolen

TRUE Random Numbers

- True random numbers generally require a physical process
- Common circuit is an unusable ring oscillator built into the CPU
  - It is then sampled at a low rate to generate true random bits which are then fed into a pRNG on the CPU
- Other common sources are human activity measured at very fine time scales
  - Keystroke timing, mouse movements, etc
    - "Wiggle the mouse to generate entropy for a key"
  - Network/disk activity which is often human driven
- More exotic ones are possible:
  - Cloudflare has a wall of lava lamps that are recorded by a HD video camera which views the lamps through a rotating prism: It is just one source of the randomness

Combining Entropy

- The general procedure is to combine various sources of entropy
- The goal is to be able to take multiple crappy sources of entropy
  - Measured in how many bits:
    A single flip of a true random coin is 1 bit of entropy
  - And combine into a value where the entropy is the minimum of the sum of all entropy sources (maxed out by the # of bits in the hash function itself)
  - **N-1** bad sources and **1** good source -> good pRNG state

Pseudo Random Number Generators (aka Deterministic Random Bit Generators)

- Unfortunately one needs a ***lot*** of random numbers in cryptography
  - More than one can generally get by just using the physical entropy source
- Enter the pRNG or DRBG
  - If one knows the state it is ***entirely predictable***
  - If one doesn't know the state it should be ***indistinguishable*** from a random string
- Three operations
  - Instantiate: (aka Seed) Set the internal state based on the real entropy sources
  - Reseed: Update the internal state based on both the previous state and ***additional entropy***
    - The big different from a simple stream cipher
  - Generate: Generate a series of random bits based on the internal state
    - Generate can also optionally add in additional entropy
- **instantiate(entropy)**
reseed(entropy)
generate(bits, {optional entropy})

Properties for the pRNG

- Can a pRNG be truly random?
  - No. For seed length **s**, it can only generate at most **2^s** distinct possible sequences.
- A cryptographically strong pRNG "looks" truly random to an attacker
  - Attacker ***cannot distinguish*** it from a random sequence:
    If the attacker can tell a sufficiently long bitstream was generated by the pRNG instead of a truly random source it isn't a good pRNG

Prediction and Rollback Resistance

- A pRNG should be predictable only if you know the internal state
  - It is this predictability which is why its called "pseudo"
- If the attacker does not know the internal state
  - The attacker should not be able to distinguish a truly random string from one generated by the pRNG
- It should also be rollback-resistant
  - Even if the attacker finds out the state at time T, they should not be able to determine what the state was at T-1
  - More precisely, if presented with two random strings, one truly random and one generated by the pRNG at time T-1, the attacker should not be able to distinguish between the two
- Not all pRNGs have rollback resistance:
  it isn't ***technically*** required of a pRNG.
  EG, CTR mode with a random key doesn’t have rollback resistance

Why "Rollback Resistance" is Essential

- Assume attacker, at time T, is able to obtain all the internal state of the pRNG
  - How? E.g. the pRNG screwed up and instead of an IV, released the internal state, or the pRNG is bad...
- Attacker observes how the pRNG was used
  - T_{-1} = Session key
    T_0 = Nonce
- Now if the pRNG doesn't resist rollback, and the attacker gets the state at T_0, attacker can know the session key! And we are back to...

More on Seeding and Reseeding

- Seeding should take all the different physical entropy sources available
  - If one source has 0 entropy, it ***must not*** reduce the entropy of the seed
  - We can shove a whole bunch of low-entropy sources together and create a high-entropy seed
- Reseeding ***adds*** in even more entropy
  - **F(internal_state, new material)**
  - Again, even if reseeding with 0 entropy, it ***must not*** reduce the entropy of the seed

Probably the best pRNG/DRBG: HMAC_DRBG

- Generally believed to be the best
  - ***Accept no substitutes***!
- Two internal state registers, ***V*** and ***K***
  - Each the same size as the hash function's output
- ***V*** is used as (part of) the data input into HMAC, while ***K*** is the key
- If you can break this pRNG you can ***either break the underlying hash function*** or ***break a significant assumption about how HMAC works***
  - Yes, security proofs sometimes are a very good thing and actually do work

HMAC_DRBG Update

- Used for both instantiate (**state.k = state.v = 0**) and reseed (keep **state.k** and **state.v**)
- Designed so that even if the attacker controls the input but doesn't know **k**: The attacker should not be able to predict the new **k**

```
function hmac_drbg_update (state, input) {
  state.k = hmac(state.k, state.v || 0x00
                          || input)
  state.v = hmac(state.k, state.v)
  state.k = hmac(state.k, state.v || 0x01
                          || input)
  state.v = hmac(state.k, state.v)
}
```

HMAC_DRBG Generate

- The basic generation function
- Remarks:
  - It requires one HMAC call per blocksize-bits of state
  - Then two more HMAC calls to update the internal state
- Prediction resistance:
  - If you can distinguish new **K** from random when you don't know old **K**:
    You've distinguished HMAC from a random function!
    Which means you've either broken the hash or the HMAC construction
- Rollback resistance:
  - If you can learn old **K** from new **K** and **V**:
    You've reversed the hash function!

```
function hmac_drbg_generate (state, n, input)
{
  tmp = ""
  while(len(tmp) < N){
     state.v = hmac(state.k,state.v)
     tmp = tmp || state.v
  }
  if input == null {
  // Update state with no input
    state.k = hmac(state.k, state.v || 0x00)
    state.v = hmac(state.k, state.v)
  } else {
    hmac_drbg_update(state, input);
  }
  // Return the first N bits of tmp
  return tmp[0:N]
}
```

UUID: Universally Unique Identifiers

- You got to have a "name" for something...
  - EG, to store a location in a filesystem
- Your name ***must*** be unique...
  - And your name ***must*** be unpredictable!
- Just chose a ***random*** value!
  - UUID: just chose a 128b random value
    - Well, it ends up being a 122b random value with some signaling information
  - A good UUID library uses a cryptographically-secure pRNG that is properly seeded
- Often written out in hex as:
  - 00112233-4455-6677-8899-aabbccddeeff

What Happens When The Random Numbers Goes Wrong...

- Insufficient Entropy:
  - Random number generator is seeded without enough entropy
- Debian OpenSSL CVE-2008-0166
  - In "cleaning up" OpenSSL (Debian 'bug' #363516), the author 'fixed' how OpenSSL seeds random numbers
    - Because the code, as written, caused Purify and Valgrind to complain about reading uninitialized memory
  - Unfortunate cleanup reduced the pRNG's seed to be ***just*** the process ID
    - So the pRNG would only start at one of ~30,000 starting points
- This made it easy to find private keys
  - Simply set to each possible starting point and generate a few private keys
  - See if you then find the corresponding public keys anywhere on the Internet

http://blog.dieweltistgarnichtso.net/Caprica,-2-years-ago

And Now Lets
Add Some RNG Sabotage...

- The Dual_EC_DRBG
  - A pRNG pushed by the NSA behind the scenes based on Elliptic Curves
- It relies on two parameters, ***P*** and ***Q*** on an elliptic curve
  - The person who generates ***P*** and selects ***Q=eP*** can predict the random number generator, regardless of the internal state
- It also ***sucked!***
  - It was horribly slow and even had subtle biases that shouldn't exist in a pRNG:
    You could distinguish the upper bits from random!
- Now this was spotted fairly early on...
  - Why should anyone use such a horrible random number generator?

Well, anyone not paid that is...

- RSA Data Security accepted ~~30 pieces of silver~~ $10M from the NSA to implement Dual_EC in th RSA BSAFE library
  - And ***silently*** make it the default pRNG
- Using RSA's support, it became a NIST standar
  - And inserted into other products...
- And then the Snowden revelations
  - The initial discussion of this sabotage in the NY Times just vaguely referred to a Crypto talk given by Microsoft people...
    - That everybody quickly realized referred to Dual_EC

But this is insanely powerful...

- It isn't just forward prediction but being able to run the generator backwards!
  - Which is why Dual_EC is so nasty:
    Even if you know the internal state of HMAC_DRBG it has rollback resistance!
- In TLS (HTTPS) and Virtual Private Networks you have a motif of:
  - Generate a random session key
  - Generate some other random data that's ***public visible***
    - EG, the IV in the encrypted channel, or the "random" nonce in TLS
    - Oh, and an NSA sponsored "standard" to spit out even more "random" bits!
- If you can run the random number generator ***backwards***, you can find the session key

It Got Worse: Sabotaging Juniper

- Juniper also used Dual_EC in their Virtual Private Networks
  - "But we did it safely, we used a different **Q**"
- Sometime later, someone else noticed this...
  - "Hmm, **P** and **Q** are the keys to the backdoor...
    Lets just hack Juniper and rekey the lock!"
    - And whoever put in the first Dual_EC then went "Oh crap, we got locked out but we can't do anything about it!"
- Sometime later, someone else goes...
  - "Hey, lets add an ssh backdoor"
- Sometime later, Juniper goes
  - "Whoops, someone added an ssh backdoor, lets see what else got F'ed with, oh, this # in the pRNG"
- And then everyone else went
  - "Ohh, patch for a backdoor. Lets see what got fixed.
    Oh, these look like Dual_EC parameters..."

Sabotaging "Magic Numbers" In General

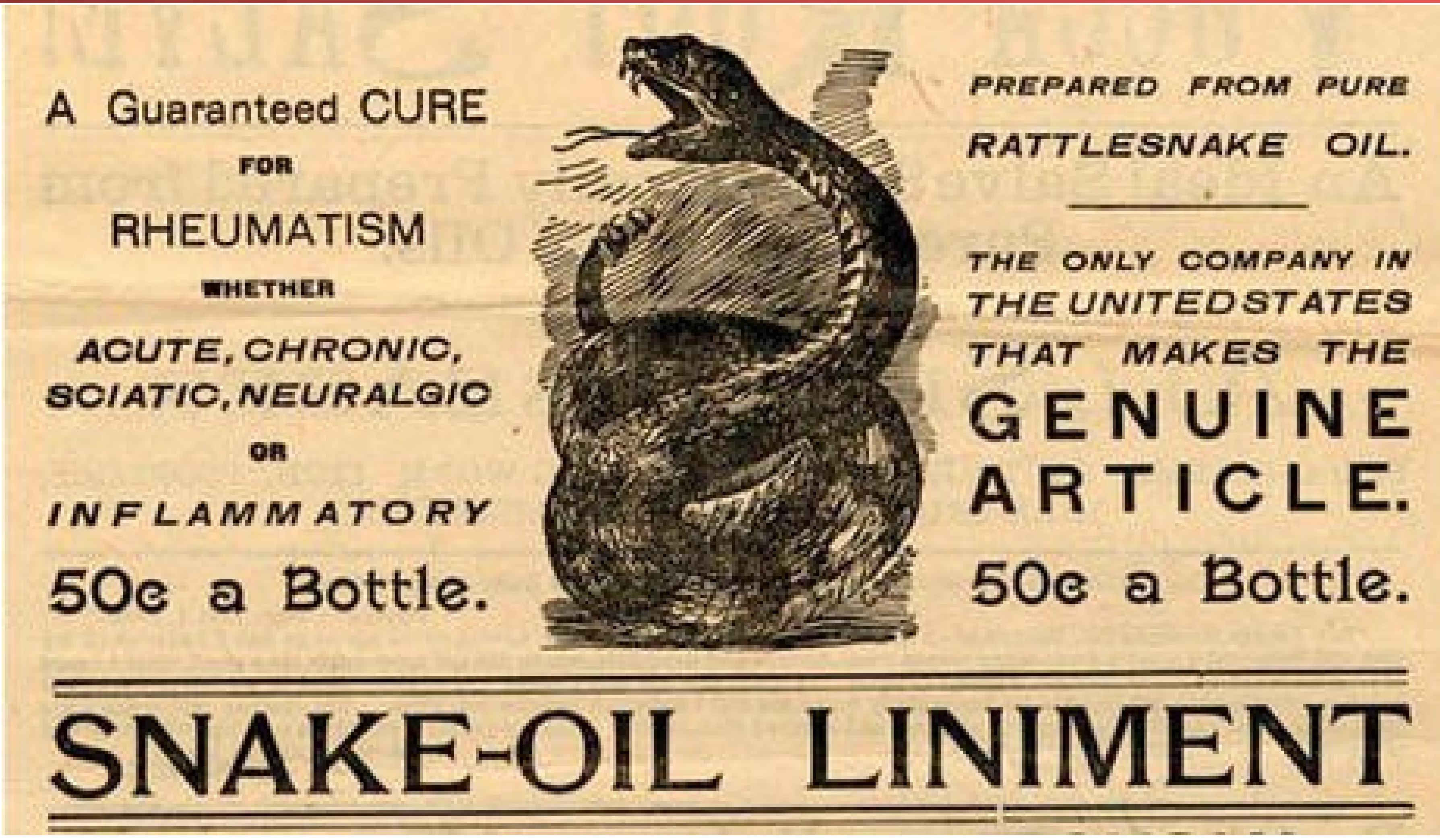
- Many cryptographic implementations depend on "magic" numbers
  - Parameters of an Elliptic curve
  - Magic points like ***P*** and ***Q***
  - Particular prime ***p*** for Diffie/Hellman
  - The content of S-boxes in block cyphers
- Good systems should cleanly describe how they are generated
  - In some sound manner (e.g. AES's S-boxes)
  - In some "random" manner defined by a pRNG with a specific seed

Because Otherwise You Have Trouble...

- Not only Dual-EC's ***P*** and ***Q***
- Recent work: 1024b Diffie/Hellman moderately impractical...
  - But you can create a sabotaged prime that is 1/1,000,000 the work to crack! And the most often used "example" ***p***'s origin is lost in time!
- It can cast doubt ***even when a design is solid***:
  - The DES standard was developed by IBM but with input from the NSA
    - Everyone was suspicious about the NSA tampering with the S-boxes...
    - They did: The NSA made them ***stronger*** against an attack they knew but the public didn't
  - The NSA-defined elliptic curves P-256 and P-384

Snake Oil Cryptography: Craptography

- "Snake Oil" refers to 19th century fraudulent "cures"
  - Promises to cure practically every ailment
  - Sold because there was no regulation and no way for the buyers to know



- The security field is practically ***full*** of Snake Oil Security and Snake Oil Cryptography
  - https://www.schneier.com/crypto-gram/archives/1999/0215.html#snakeoil

Anti-Snake Oil: NSA's CNSA cryptographic suite

- Successor to "Suite B"
  - Unclassified algorithms approved for Top Secret:
    - There is nothing higher than TS, you have "compartments" but those are access control modifiers
    - https://www.iad.gov/iad/programs/iad-initiatives/cnsa-suite.cfm
  - Symmetric key, AES: 256b keys
  - Hashing, SHA-384
  - RSA/Diffie Helman: >= 3072b keys
  - ECDHE/ECDSA: 384b keys over curve P-384
- In an ideal world, I'd only use those parameters,
  - But a lot of "strong" commercial is 128b AES, SHA-256, 2048b RSA/DH, 256b elliptic curves, plus the DJB curves and cyphers (ChaCha20)
  - NSA has a requirement where a Top Secret communication captured today should not be decryptable by an adversary 40 years from now!

Snake Oil Warning Signs...

- Amazingly long key lengths
  - The NSA is super paranoid, and even they don't use >256b keys for symmetric key or >4096b for RSA/DH public key
  - So if a system claims super long keys, be suspicious
- New algorithms and crazy protocols
  - There is ***no reason*** to use a novel block cipher, hash, public key algorithm, or protocol
    - Even a "post quantum" public key algorithm should not be used alone:
Combine it with a conventional public key algorithm
  - Anyone who roles their own is asking for trouble!
  - EG, Telegram
    - "It's like someone who had never seen cake but heard it described tried to bake one.
With thumbtacks and iron filings." Matthew D Green
    - "Exactly! GLaDOS-cake encryption.
Odd ingredients; strange recipe; probably not tasty; may explode oven. :)" Alyssa Rowan

Lots in the Cryptocurrency Space…

- The biggest being IOTA (aka IdiOTA), a “internet of Things” cryptocurrency…
  - That doesn’t use public key signatures, instead a hash based scheme that means you can ***never*** reuse a key…
    - And results in 10kB+ signatures! (Compared with RSA which is <450B, and those are big)
  - That has created their own hash function…
    - That was quickly broken!
  - That is supposed to end up distributed…
    - But relies entirely on their central authority
  - That uses ***trinary math!?!***
    - Somehow claiming it is going to be better, but you need entirely new processors…

Snake Oil Warning Signs...

- "One Time Pads"
  - One time pads are secure, if you actually have a true one time pad
  - But almost all the snake oil advertising it as a "one time pad" isn't!
  - Instead, they are invariably some wacky stream cypher
- Gobbledygook, new math, and "chaos"
  - Kinda obvious, but such things are never a good sign
- Rigged "cracking contests"
  - Usually "decrypt this message" with no context and no structure
    - Almost invariably a single or a few unknown plaintexts with nothing else
  - Again, Telegram, I'm looking at you here!

Unusability:
No Public Keys

- The APCO Project 25 radio protocol
  - Supports encryption on each traffic group
    - But each traffic group uses a single ***shared*** key
- All fine and good if you set everything up at once...
  - You just load the same key into all the radios
  - But this totally fails in practice: what happens when you need to coordina somebody else who doesn't have the same keys?
- Made worse by bad user interface and users who think r frequently is a good idea
  - If your crypto is good, you shouldn't need to change your crypto keys
- "Why (Special Agent) Johnny (Still) Can't Encrypt
  - http://www.crypto.com/blog/p25

Unusability: PGP

- I ***hate*** Pretty Good Privacy
  - But not because of the cryptography...
- The PGP cryptography is decent...
  - Except it lacks "Forward Secrecy":
    If I can get someone's private key I can decrypt all their old messages
- The metadata is awful...
  - By default, PGP says who every message is from and to
    - It makes it much faster to decrypt
  - It is hard to hide metadata well, but its easy to do things better than what PGP does
- It is never transparent
  - Even with a "good" client like GPG-tools on the Mac
  - And I don't have a client on my cellphone

Unusability:
How do you find someone's PGP key?

- Go to their personal website?
- Check their personal email?
- Ask them to mail it to you
  - In an unencrypted channel?
- Check on the MIT keyserver?
  - And get the old key that was mistakenly uploaded and can never be removed?

Search results for 'nweaver icsi edu berkeley'

```
Type bits/keyID     Date       User ID
-----------------------------------------------------------------------
pub  4096R/8A46A420 2013-06-20 Nicholas Weaver <nweaver@icsi.berkeley.edu>
                               Nicholas Weaver <n_weaver@mac.com>
                               Nicholas Weaver <nweaver@gmail.com>
-----------------------------------------------------------------------
pub  2048R/442CF948 2013-06-20 Nicholas Weaver <nweaver@icsi.berkeley.edu>
```