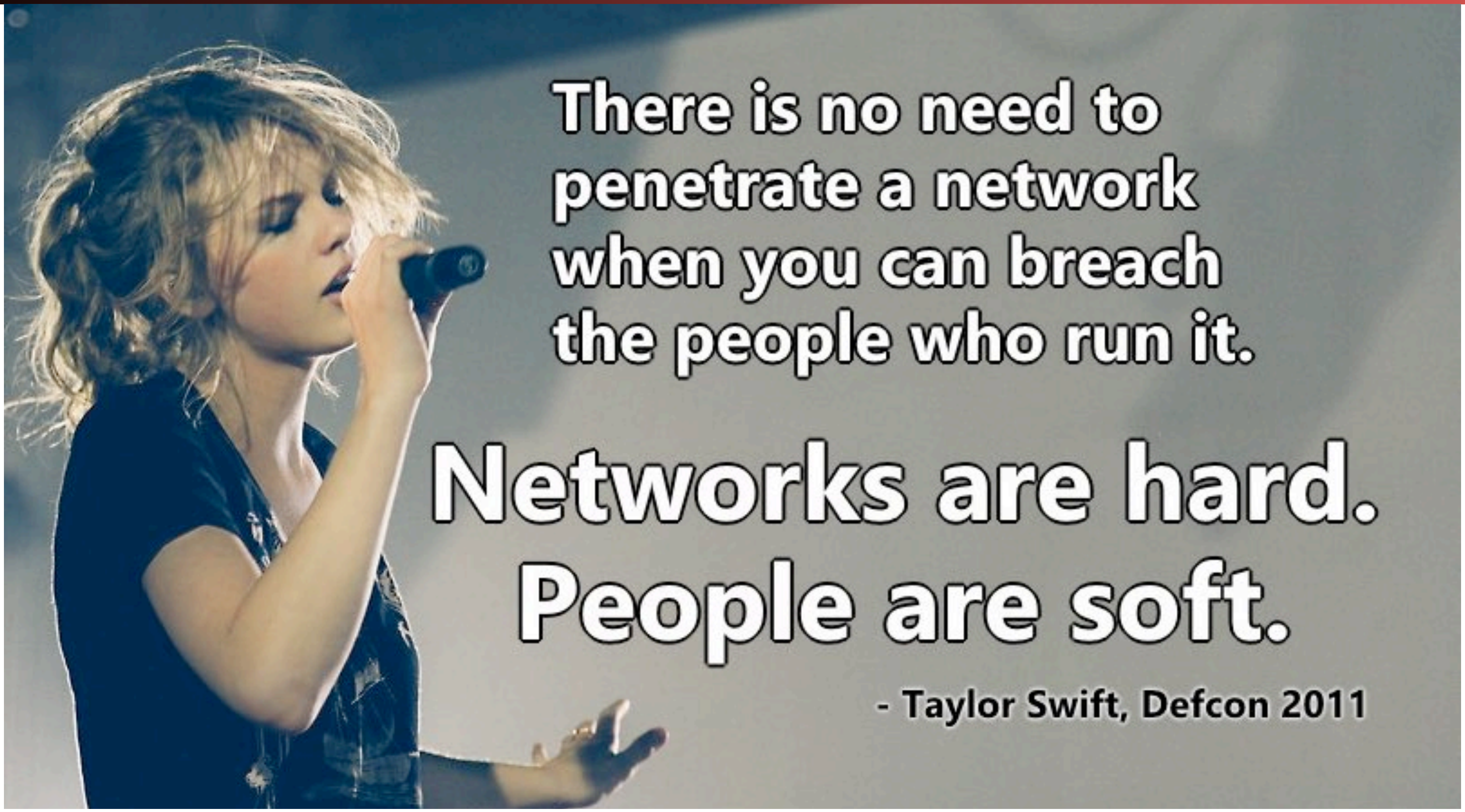
# Applied Cryptography
# Applied Craptography
# Network Security

# Meme of the Day





# Outline



- Applied Cryptography
  - HMAC
  - Facebook Messenger Abuse Complaints
  - Generating random numbers
- Applied Craptography
  - Snake Oil
  - Unusable systems
  - Low entropy RNGs
  - Sabotaged RNGs
  - Sabotaged "Magic Numbers"
- Network Security
  - Introduction and Motivation

# Another MAC construction: HMAC



- Idea is to turn a hash function into a MAC
  - Since hash functions are often much faster than encryption
  - While still maintaining the properties of being a cryptographic hash
- XOR the key with the i_pad
  - 0x363636... (one hash block long)
- Hash ((K ⊕ i_pad) || message)
- XOR the key with the o_pad
  - 0x5c5c5c...
- Hash ((K ⊕ o_pad) || first hash)

```
function hmac (key, message) {
    if (length(key) > blocksize) {
        key = hash(key)
    }
    while (length(key) < blocksize) {
       key = key || 0x00
    }
    o_key_pad = 0x5c5c... ⊕ key
    i_key_pad = 0x3636... ⊕ key
    return hash(o_key_pad ||
                hash(i_key_pad || message))
}
```

# Why This Structure?



- i_pad and o_pad are slightly arbitrary
  - But it is necessary for security for the two values to be different
    - So for paranoia chose very different bit patterns
- Second hash prevents appending data
  - Otherwise attacker could add more to the message and the HMAC and it would still be a valid HMAC for the key
    - Wouldn't be a problem with the key at the ***end*** but at the start makes it easier to capture intermediate HMACs
- Is a Pseudo Random Function if the underlying hash is a PRF

```
function hmac (key, message) {
    if (length(key) > blocksize) {
        key = hash(key)
    }
    while (length(key) < blocksize) {
       key = key || 0x00
    }
    o_key_pad = 0x5c5c... ⊕ key
    i_key_pad = 0x3636... ⊕ key
    return hash(o_key_pad ||
                hash(i_key_pad || message))
}
```

# The Facebook Problem: Applied Cryptography in Action



- Facebook Messenger now has an encrypted chat option
  - Limited to their phone application
- The cryptography in general is very good but uninteresting
  - Used a well regarded asynchronous messenger library (from Signal) with many good properties
- When Alice wants to send a message to Bob
  - Queries for Bob's public key from Facebook's server
  - Encrypts message and send it to Facebook
  - Facebook then forwards the message to Bob
- Both Alice and Bob are using encrypted and authenticated channels to Facebook

# Facebook's Unique Messenger Problem: Abuse



- Much of Facebook's biggest problem is dealing with abuse...
  - What if either Alice or Bob is a stalker, an a-hole, or otherwise problematic?
    - Aside: A huge amount of abuse is explicitly gender based, so I'm going to use "Alex" as the abuser and "Bailey" as the victim through the rest of this example
- Facebook would expect the other side to complain
  - And then perhaps Facebook would kick off the perpetrator for violating Facebook's Terms of Service
- But fake abuse complaints are also a problem
  - So can't just take them on face value
- And abusers might also want to release info publicly
  - Want sender to be able to deny to the public but not to Facebook

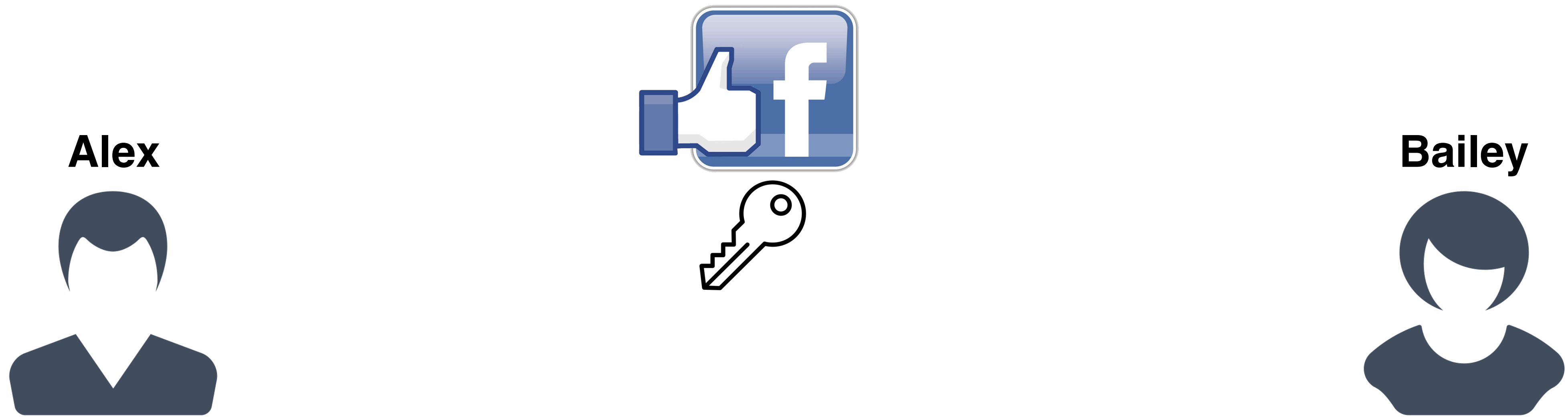
# Facebook's Problem Quantified



- Unless Bailey forwards the unencrypted message to Facebook
  - Facebook ***must not*** be able to see the contents of the message
- If Bailey does forward the unencrypted message to Facebook
  - Facebook ***must ensure*** that the message is what Alex sent to Bailey
- Nobody ***but*** Facebook should be able to verify this: No public signatures!
  - Critical to prevent abusive release of messages to the public being verifiable

# The Protocol In Action






What Is Bailey's Public Key?

# Aside: Key Transparency...



- Both Alex and Bailey are trusting Facebook's honesty...
  - What if Facebook gave Alex a different key for Bailey? How would he know?
- Facebook messenger has a ***nearly*** hidden option which allows Alex to see Bailey's key
  - If they ever get together, they can manually verify that Facebook was honest
- The mantra of central key servers: ***Trust but Verify***
  - The simple option is enough to force honesty, as each attempt to lie has some probability of being caught
- This is the biggest weakness of Apple iMessage:
  - iMessage has (fairly) good cryptography but there is no way to verify Apple's honesty

# The Protocol In Action



**Alex**

{message=E($K_{pub\_b}$,
M={"Hey Bailey I'm going to
say something abusive",
$k_{rand}$}),
mac=HMAC($k_{rand}$, M),
to=Bailey}

{message=E($K_{pub\_b}$,
M={"Hey Bailey I'm going to
say something abusive",
$k_{rand}$}),
mac=HMAC($k_{rand}$, M),
to=Bailey,
from=Alex,
time=now,
fbmac=HMAC($K_{fb}$,{mac, from,
to, time})}

**Bailey**

# Some Notes



- Facebook ***can not*** read the message or even verify Alex's HMAC
  - As the key for the HMAC is in the message itself
- Only Facebook knows their HMAC key
  - And its the only information Facebook ***needs*** to retain in this protocol: Everything else can be discarded
- Bailey upon receipt checks that Alex's HMAC is correct
  - Otherwise Bailey's messenger silently rejects the message
    - Forces Alex's messenger to be honest about the HMAC, even thought Facebook never verified it

# Now To
# Report Abuse



**Alex**

**Bailey**

```
{Abuse{
  M={"Hey Bailey I'm going to
     say something abusive",
     k_rand}},
 mac=HMAC(k_rand, M),
 to=Bailey,
 from=Alex,
 time=now,
 fbmac=HMAC(K_fb,{mac, from,
                  to, time})}
```

# Facebook's Verification



- First verify that Bailey correctly reported the message sent
  - Verify **`fbmac=HMAC(`**$K_{fb}$**`,{mac,from,to,time})`**
    - Only Facebook can do this verification since they keep $K_{fb}$ secret
  - This enables Facebook to confirm that this is the message that it relayed from Alex to Bailey
- Then verify that Bailey didn't tamper with the message
  - Verify **`mac=HMAC(`**$k_{rand}$**`,{M, `**$k_{rand}$**`})`**
- Now Facebook knows this was sent from Alex to Bailey and can act accordingly
  - But Bailey can't prove that Alex sent this message to anyone other than Facebook
  - And Bailey can't tamper with the message because the HMAC is also a hash

# Random Number Generators



- The Random Number Generator is the heart of cryptography
- It gets used all the time
  - "Select a ***random*** a..." in your Diffie/Hellman key exchange
  - "Create a ***random*** k..." for the session key
  - "Create a ***random*** k..." for the HMAC key in the previous protocol
- But true random numbers are very hard to get
  - Especially in large amounts
- Result is "gather entropy and use a pseudo random number generator"

# *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
- 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

# Combining Entropy



- The general procedure is to combine various sources of entropy
  - Usually using a hash function
- The goal is to be able to take multiple crappy sources of entropy
  - Measured in how many bits:
    A single flip of a 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)

# 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
  - Generate: Generate a series of random bits based on the internal state
    - Generate can also optionally add in additional entropy

# 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
  - 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
  - This is essential:
    - A common motif: Generate a random session key, then do something else that involves some "random" values but which an attacker might see

# Probably the best pRNG/DRBG: HMAC_DRBG



- Generally believed to be the best
  - Breaking it requires either breaking the particular hash function or breaking the assumption that HMAC is distinguishable from random
- 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

# 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
- Backtrack resistance:
  - If you can learn old K from new K and V:
    You've reversed the hash function!
- 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

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

# HMAC_DRBG Update



- Used instead of the "no-input update" when you have additional entropy on the generate call
- Used standalone for both instantiate (state.k = state.v = 0) and reseed
- 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)
}
```

# Now Onto The Craptography...

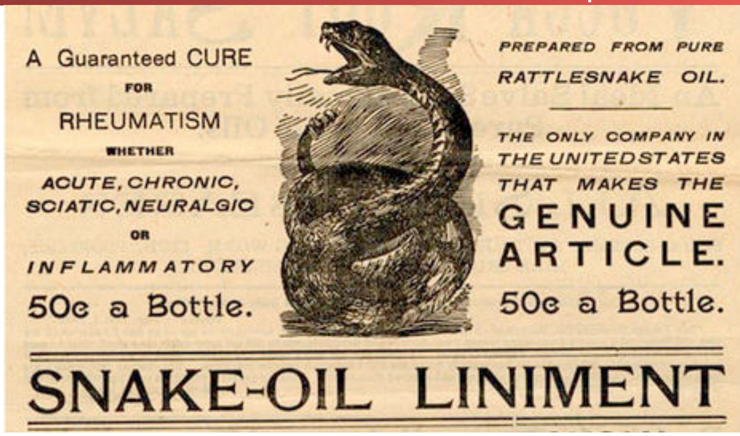


- Snake Oil
- Unusable systems
- Low entropy RNGs
- Sabotaged RNGs
- Sabotaged "Magic Numbers"

# Snake Oil Cryptography



- "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//Sensitive Compartmented Information
    - 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

# 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 coordinate with somebody else who doesn't have the same keys?
- Made worse by bad user interface and users who think rekeying 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><br>Nicholas Weaver <n_weaver@mac.com><br>Nicholas Weaver <nweaver@gmail.com> |
| pub | 2048R/442CF948 | 2013-06-20 | Nicholas Weaver <nweaver@icsi.berkeley.edu> |

# Unusable:
# openssl libcrypto and libssl



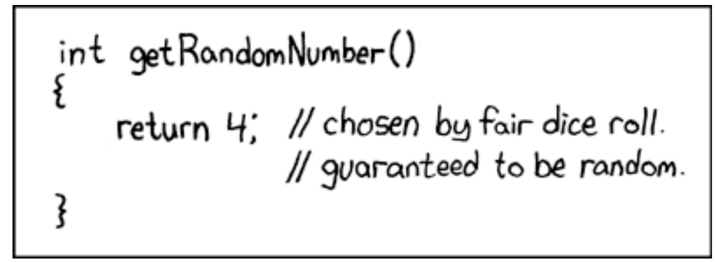
- OpenSSL is a nightmare...
  - A gazillion different little functions needed to do anything
- So much of a nightmare that I'm not going to bother learning it to teach you how bad it is
  - This is why we didn't give you pycrypto raw, but instead provided a wrapper in the project
- But just to give you an idea:
The command line OpenSSL utility options:

```
OpenSSL> help
openssl:Error: 'help' is an invalid command.

Standard commands
asn1parse         ca                ciphers           cms
crl               crl2pkcs7         dgst              dh
dhparam           dsa               dsaparam          ec
ecparam           enc               engine            errstr
gendh             gendsa            genpkey           genrsa
nseq              ocsp              passwd            pkcs12
pkcs7             pkcs8             pkey              pkeyparam
pkeyutl           prime             rand              req
rsa               rsautl            s_client          s_server
s_time            sess_id           smime             speed
spkac             srp               ts                verify
version           x509

Message Digest commands (see the `dgst' command for more details)
md4               md5               mdc2              rmd160
sha               sha1

Cipher commands (see the `enc' command for more details)
aes-128-cbc       aes-128-ecb       aes-192-cbc       aes-192-ecb
aes-256-cbc       aes-256-ecb       base64            bf
bf-cbc            bf-cfb            bf-ecb            bf-ofb
camellia-128-cbc  camellia-128-ecb  camellia-192-cbc  camellia-192-ecb
camellia-256-cbc  camellia-256-ecb  cast              cast-cbc
cast5-cbc         cast5-cfb         cast5-ecb         cast5-ofb
des               des-cbc           des-cfb           des-ecb
des-ede           des-ede-cbc       des-ede-cfb       des-ede-ofb
des-ede3          des-ede3-cbc      des-ede3-cfb      des-ede3-ofb
des-ofb           des3              desx              idea
idea-cbc          idea-cfb          idea-ecb          idea-ofb
rc2               rc2-40-cbc        rc2-64-cbc        rc2-cbc
rc2-cfb           rc2-ecb           rc2-ofb           rc4
rc4-40            seed              seed-cbc          seed-cfb
seed-ecb          seed-ofb          zlib
```

# Some Protocols Are Especially Vulnerable to Reuse



- El-Gamal, DSA, and ECDSA algorithms very vulnerable to value reuse
  - Most famous is actually Sony PS3:
    It contained a special key LV0 used to decrypt the firmware
  - The algorithms all use a random value k
    - If you ever repeat k, or even use a ***predictable*** k, you are sunk...
- Sony signed in multiple places with the same k
  - Enabled determining all their private keys! OOPS




From XKCD

# 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 to implement Dual_EC in their RSA BSAFE library
  - And silently make it the default pRNG
- Using RSA's support, it became a NIST standard
  - 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!
- 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
- 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!
- It can cast doubt ***even when a design is solid***:
  - The DES standard developed by IBM 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
    - I trust them because they are in Suite-B/CNSA so the NSA uses them for TS communication:
      A backdoor here would be absolutely unacceptable

# Shifting Gears: Network Security



- Networking (CS168)
  - Lets take this unreliable communication mechanism and make something useful out of it
- Network Security
  - Lets take this unreliable and insecure communication mechanism and make something useful and secure out of it
    - It unfortunately means networking becomes a prerequisite for security...
- Generally takes two forms
  - Hacks that attempt to prevent deficiencies
  - Using encrypted protocols to make the layers underneath irrelevant
- My plan: Incremental concepts
  - I'm going to start at the "bottom" and work up, discussing functionality and security problems together

# The OSI 7 Layer Network Stack



- Physical and Data Link:
  - Ethernet and Wireless Ethernet
  - DHCP and ARP
- Network Layer:
  - IP
  - DNS
- Transport Layer:
  - TCP and UDP
  - TLS
  - Firewalls
- Application Layer:
  - Network Intrusion Detection
  - Leads into Web Security

| |
|---|
| **Political Layer** |
| **Application Layer** |
| **Presentation Layer** |
| **Session Layer** |
| **Transport Layer** |
| **Network Layer** |
| **Data Link Layer** |
| **Physical Layer** |