

# Crypto Failures

## From basics to advanced stuff

Guy Barnhart-Magen  
DC9723



# Who am I?

Father of two, hacker

**BSidesTLV** chairman and CTF member

(Lucky to speak at many conferences)

Today: **SecureAI** CTO

Before: Intel, Cisco and a couple of Startups

OS Hardening, Crypto, Embedded Security, Security of ML

**@barnhartguy**



OWASP  
AppSec Israel 2018



CRYPTO + PRIVACY  
VILLAGE



# Agenda

New format

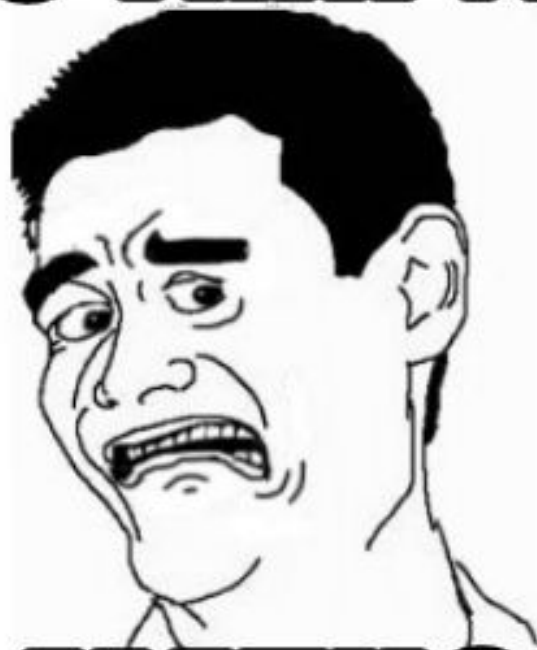
Two Parts

- Intro to Crypto
  - Some failures, mostly engineering

## Break

- Advanced stuff
  - Multi Party Computation
  - CurveBall

**IS THAT...**



**MATH?**

# Building Blocks



# BASICS

**Confidentiality**

Keeping things secret

**Integrity**

Can we forge a message?

**Authenticity**

Can we trust that this is from the sender?

# What do we need to build a secure protocol?

Hash Functions

Encryption

Signatures

Randomness

# How Much Time to Break?

Linear Time

Polynomial Time

Exponential Time



# Trap Door Problems

Known parameters  $\Rightarrow$  computation easy

Unknown parameters  $\Rightarrow$  very hard

Once mixed, difficult to unmix



# Cryptographic Hash Functions

One way function

1 bit change in input  
changes ~50% of output

Looks just like noise

```
SHA1("The quick brown fox jumps over  
the lazy  
dog") 2fd4e1c67a2d28fced849ee1bb76e73  
91b93eb12
```

```
SHA1("The quick brown fox jumps over  
the lazy  
cog") de9f2c7fd25e1b3afad3e85a0bd17d9  
b100db4b3
```

$2^{80}$  to bruteforce, 2019 attack:  $2^{68}$

This is now practical with ~\$100K of Amazon EC2

# Symmetric Cryptography

Two way function

1 bit change in input  
changes ~50% of output

Looks just like noise

Very fast

```
msg = aes.encrypt("The quick brown fox  
jumps over the lazy dog" +  
"\0"*21)caf5f61978f250e4f34533fcba2ffe54  
1623aa5c6be805c27df12a00659a2d6857138f7e  
efcf4fa09d301200091cec8fc30614ad5e9ae2e2  
740f3bcc550468f6
```

```
msg = aes.encrypt("The quick brown fox  
jumps over the lazy cog" +  
"\0"*21)272e9f8c4d8b7e52800c9dddfdb96aa2  
2fc0a28f69aa2ef90ce580d34b3bd29c9de87c59  
859e70e30cd050721bd28787c92cb54b47fc3b47  
5c78d3731832d74d
```

# Asymmetric Cryptography

Different operations to encrypt and  
decrypt

Very Slow

Create a Private and Public key pair (keep your private key secret)

```
PubKey, PrvKey = GenASymmKeys(RSA, 4096)
```

Give everyone your public key

Now anyone can create a message, encrypt it with your public key

```
EncMsg = Enc(msg, PubKey)
```

Only you can decrypt it with `PrvKey`

```
msg = Dec(EncMsg, PrvKey)
```

# Signatures

Prove your identity by proving you have  
a secret tied to your public identity

Very Slow

Having a `PrvKey` is equivalent to an identity  
(holding the key is proof you have the secret)

```
h = hash(msg)
signature = EncMsg(h, PrvKey)
```

Now anyone can verify that you signed `msg`, with  
your published `PubKey`

```
h = hash(msg)
h` = DecMsg(signature, PubKey)
h == h` proves you signed msg
```

# Randomness

The chance to predict the next bit is 50%

TRNG - True Random Number Generator

PRNG - Pseudo Random Number Generator

`random=PRNG (seed=TRNG)`

What happens when the random isn't random?

We often assume `nonce` values are random

# RSA

Reversing exponents...

A dark blue diagonal gradient bar that starts from the bottom left and extends towards the top right, covering the lower half of the slide.



# A Little Math

$$X^2 * X^3 = X^5$$

$$13 \bmod 12 = 1$$

# RSA or the Discrete Log Problem

1.  $n = p * q$  (prime numbers)
2. Choose  $e$  (e.g.  $2^{16} + 1 = 65537$ )
3.  $d = e^{-1} \bmod n$   
     $\text{Pub}\{e, n\}$   
     $\text{Prv}\{d\}$

# RSA or the Discrete Log Problem

Encryption:

$$C = m^e \bmod n$$

Decryption:

$$m = C^d \bmod n = (m^e)^d \bmod n = m$$

# RSA or the Discrete Log Problem

$$P = 61, q = 53 \Rightarrow n = \mathbf{3233}$$

$$\phi(3233) = 780 \Rightarrow \mathbf{e = 17}$$

$$d * e = 1 \bmod \phi(n) \Rightarrow 413 * 17 = 1 \bmod 780 \Rightarrow \mathbf{d = 413}$$

$$\text{Pub}\{n=3233, e=17\}, \text{Prv}\{n=3233, d=413\}$$

$$C(65) = 65^{413} \bmod 3233 = 2790$$

$$m = 2790^{413} \bmod 3233 = 65$$

# Binary Exponentiation?

$$3^{13} = 3^{1101} = 3^8 * 3^4 * 3^1$$

$$3^1 = 3$$

$$3^2 = 9$$

$$3^4 = 81$$

$$3^8 = 6561$$

$$3^{13} = 6561 * 81 * 3 = 1594323$$



# Attacks

Timing

Power

Memory

Padding

Compression

Oracle

# Elliptic Curves

Finding a point on a curve...





# Bitcoin

Coin = Finding hash collisions (mining)

Wallets = Identities are tied to private keys

Transactions = Signing coin transfers from wallet to wallet

Blockchain = a decentralized database to store coins (**hash collisions**) and transactions

Wallets/Identities/**private keys** (anonymous), but not confidential - everything is recorded!

Lose your private key, you cannot **sign** transactions, lose access to those coins (frozen money)

# Elliptic Curve Cryptography (ECC)

Asymmetric

Similar to RSA

Much faster


## Wallet

`Address = Hash (PubKey)`

**1DY5YvRxSwomrK7nELDZzAidQQ6ktjR**

“This money I can spend, can now be spent by X”

# BITCOIN TRANSACTION

+ 12ff51d83eac5b69b222af02ffe68454b7d5501b6dcdf2f33fec25c8510d53fd 

mined Feb 18, 2018 10:20:47 PM

1GDFhSxaemytQhqYZarabatwpwuZoMNHAG 7.17974665 BTC



33Xis87i9P1ui5to1PoDic2LVYfMUUn2Tt2 0.06564117 BTC (U)

15QaHxet56z1ES36xSsyh8mQBE5NY97n2k 7.11144294 BTC (S)

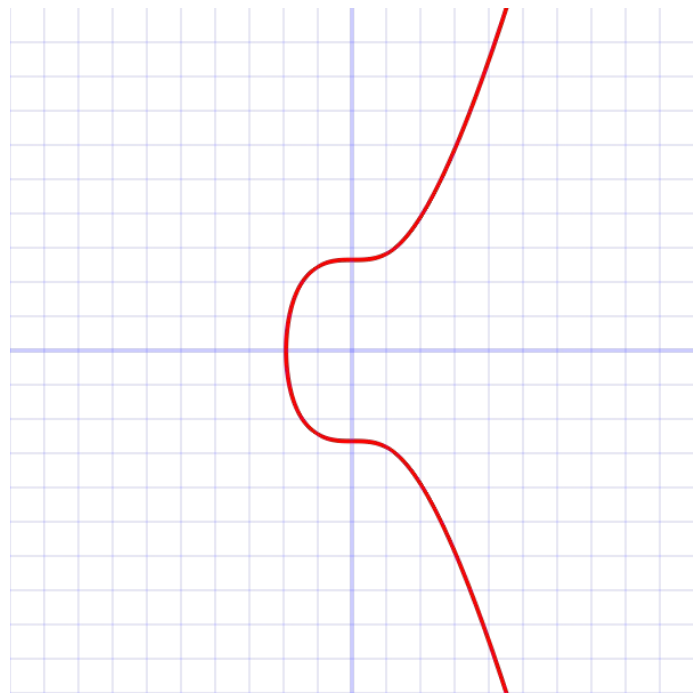
FEE: 0.00266254 BTC

1 CONFIRMATIONS

7.17708411 BTC

# Elliptic Curves

- **Base point:** point  $G$  on a curve
- **Private Key:** a random number  $d$
- **Public Key:**  $d \times G$
- $Y^2 = X^3 + 7$ 
  - But over  $\mathbb{Z}_p$
  - and modular



# Elliptic Curve Cryptography (ECC)

Known: G

h: hash of the msg

d: private key

k: random number

(r,s): signature

To create a transaction we need to sign our message

```
msg = "This money I can spend, can now be  
spent by X"
```

```
h = hash(msg)
```

```
k = random()
```

```
r = cross_op(k,G) //also random
```

```
sig = sign(h,r,k,d)
```

We publish `sig` on the blockchain, once accepted then X has the money

$$(r, s) = \left(r, \frac{h + rXd}{k}\right)$$

$$r = kXG$$

# What if $k$ isn't random?

$k$  isn't random

$r$  isn't random either

$s$  is public on the blockchain

1. Find a couple of transaction that share the same  $k$
2. Do some algebra, extract  $k$
3. Now use  $k$  and  $s$  to extract  $d$
4.  $d$  is the private key
5. ...
6. Profit!

$$(r, s) = \left( r, \frac{h + rXd}{k} \right)$$

$$r = kXG$$

# IF $k$ IS NOT RANDOM ...

- $G$  = base point of the curve
  - Known parameter
- $h$  = hash(message)
- $d$  = private key
- $k$  = a random number
- $(r, s)$  = signature
  - Published on the Blockchain
- $(r_1) = (k_1 XG)$
- $(r_2) = (k_2 XG)$
- Then...
- $r_1 = r_2$

# IF WE HAVE A COLLISION?

- $h = \text{hash}(\text{message})$
- $d = \text{private key}$
- $k = \text{a random number}$
- $(r, s) = \text{signature}$
- $r_1 = r_2$
- $s_1 = \frac{h_1 + rXd}{k}$
- $s_2 = \frac{h_2 + rXd}{k}$
- Then ...
- $s_1 - s_2 = \frac{h_1 + rXd}{k} - \frac{h_2 + rXd}{k}$
- $s_1 - s_2 = \frac{h_1 - h_2}{k} \Rightarrow k = \frac{h_1 - h_2}{s_1 - s_2}$




# IF WE KNOW $k$ ?

- $h = \text{hash}(\text{message})$
  - $d = \text{private key}$
  - $k = \text{a random number}$
  - $(r, s) = \text{signature}$
- $k = \frac{h_1 - h_2}{s_1 - s_2}$
  - $s = \frac{h + rXd}{k} \Rightarrow d = \frac{s*k - h}{r}$


# AND NOW IN ENGLISH?

- $h = \text{hash}(\text{message})$
  - $d = \text{private key}$
  - $k = \text{a random number}$
  - $(r, s) = \text{signature}$
1. Find two transactions with the same  $r$  value
  2. Find the random nonce:  $k = \frac{h_1 - h_2}{s_1 - s_2}$
  3. Find the private key:  $d = \frac{kXs - h}{r}$
  4. Sign transactions yourself
  5. ...
  6. Profit!




But this cannot happen in  
the real world!


A dark blue, diagonal shape that starts from the bottom left corner and extends towards the top right, covering the lower half of the slide. It has a smooth, curved edge.

# THIS CAN'T HAPPEN IN THE REAL WORLD!

 Author

Topic: Bad signatures leading to 55.82152538 BTC theft (so far) (Read 33421 times)

**BurtW**  
Legendary  
  
 Online  
Activity: 1218  
I no longer support vanity addresses  
  
Ignore

 **Bad signatures leading to 55.82152538 BTC theft (so far)** #1  
August 10, 2013, 10:53:13 PM

I have only seen this discussed in the newbies section so I thought I would open a thread here for a more technical discussion of this issue.

Several people have reported their BTC stolen and sent to <https://blockchain.info/address/1HKywxIL4JziqXrzLKhmB6a74ma6kxbSDj>

As you can see the address currently contains 55.82152538 stolen coins.

It has been noticed that the coins are all transferred in a few hours after a client improperly signs a transaction by reusing the same random number. As discussed here:

# THIS CAN'T HAPPEN IN THE REAL WORLD!

```
m = open("/dev
```

## Sony's ECDSA code

```
int getRandomNumber()  
{  
    return 4; // chosen by fair dice roll.  
              // guaranteed to be random.  
}
```

fail0verflow

# A small note regarding bitcoin...

Anonymity vs. Confidentiality



# Anonymity vs. Confidentiality

Anonymous - no one knows your identity

Confidential - no one knows what you're doing

Bitcoin is not Anonymous, all transactions are recorded on the blockchain

Remember WannaCry?









# Goto Fail

Chain of trust is established  
by verifying signatures, all  
the way to a trusted root

```
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;  
hashOut.length = SSL_SHA1_DIGEST_LEN;  
if ((err = SSLFreeBuffer(&hashCtx)) != 0)  
    goto fail;  
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx))  
    != 0)  
    goto fail;  
if ((err = SSLHashSHA1.update(&hashCtx,  
    &clientRandom)) != 0)  
    goto fail;  
if ((err = SSLHashSHA1.update(&hashCtx,  
    &serverRandom)) != 0)  
    goto fail;  
if ((err = SSLHashSHA1.update(&hashCtx,  
    &signedParams)) != 0)  
    goto fail;  
if ((err = SSLHashSHA1.final(&hashCtx,  
    &hashOut)) != 0)  
    goto fail;  
  
err = sslRawVerify(...);
```

# Goto Fail

Chain of trust is established  
by verifying signatures, all  
the way to a trusted root

```
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;  
hashOut.length = SSL_SHA1_DIGEST_LEN;  
if ((err = SSLFreeBuffer(&hashCtx)) != 0)  
    goto fail;  
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx))  
    != 0)  
    goto fail;  
if ((err = SSLHashSHA1.update(&hashCtx,  
    &clientRandom)) != 0)  
    goto fail;  
if ((err = SSLHashSHA1.update(&hashCtx,  
    &serverRandom)) != 0)  
    goto fail;  
if ((err = SSLHashSHA1.update(&hashCtx,  
    &signedParams)) != 0)  
    goto fail;  
    goto fail;  
if ((err = SSLHashSHA1.final(&hashCtx,  
    &hashOut)) != 0)  
    goto fail;  
  
err = sslRawVerify(...);
```

# Bitlocker

Bitlocker (SW) will use HW if available

Several self encrypting drives had weak firmware, leading to attacks

- Password and DEK not linked
- Single DEK for entire disk
- non-random DEK
- Wear leveling
- Zero buffer as password



# Writing your own crypto

Crypto is hard for experts

Never roll your own crypto!



Dinesh Singh Sikarwar  
Freelance Web Solutions

Hi Friends  
you can use it

```
function encodeString($str){  
    for($i=0; $i<5;$i++){  
    {  
        $str=strrev(base64_encode($str)); //apply base64 first and then reverse the string  
    }  
    return $str;  
}
```

return \$str;



@Dinesh: That has to be the worst example I've ever seen. Not only does it not provide any security, but it's horrendously space-inefficient. A 1024 byte message when base64-encoded becomes 1362 bytes long, when reversed and encoded again you get 1812, followed by 2412, 3208, and finally 4268 bytes.

And yet, you offer NO security. Base64 encoding accomplishes nothing, and I can safely say any hacker with an IQ above 10 will know how to crack it!

So congratulations on being the provider of the worst password storage solution to this entire 200+ "use md5" debacle.

☒ Send me an email for each new comment.

Add Comment

# CryptHook

Hook Send/Recv system calls

Uses GCM to authenticate

Encrypt with a fixed key

Same key and no sequence numbers leads to:

- Replay attack
- Message re-ordering
- Selectively dropping messages
- No session key - no forward secrecy



# WHERE TO GET ADVICE?

<https://www.reddit.com/r/crypto>

<https://crypto.stackexchange.com/>

Consultants/Experts

Crypto Review Firms

**Never roll your own  
crypto!**

# LIBRARIES YOU SHOULD LOOK AT

Always vet your own libraries

Don't trust people that roll their own crypto

Performance is a major differentiator

Which feature do you need? Attack model?

There are more options than OpenSSL

[https://en.wikipedia.org/wiki/Comparison\\_of\\_cryptography\\_libraries](https://en.wikipedia.org/wiki/Comparison_of_cryptography_libraries)

<https://github.com/sobolevn/awesome-cryptography#frameworks-and-libraries>

BoringSSL

Golang/crypto

NaCL

NSS

WolfCrypt

MatrixSSL



# WHAT CAN YOU DO?

Leave implementation of crypto algorithms to experts, **update when possible**

Take classes, courses, training and workshops - **become an expert yourself**

Read a lot, have a relevant degree (Math, EE, CS) - follow up on papers and conferences

Review your code by experts, save a lot of headache down the road

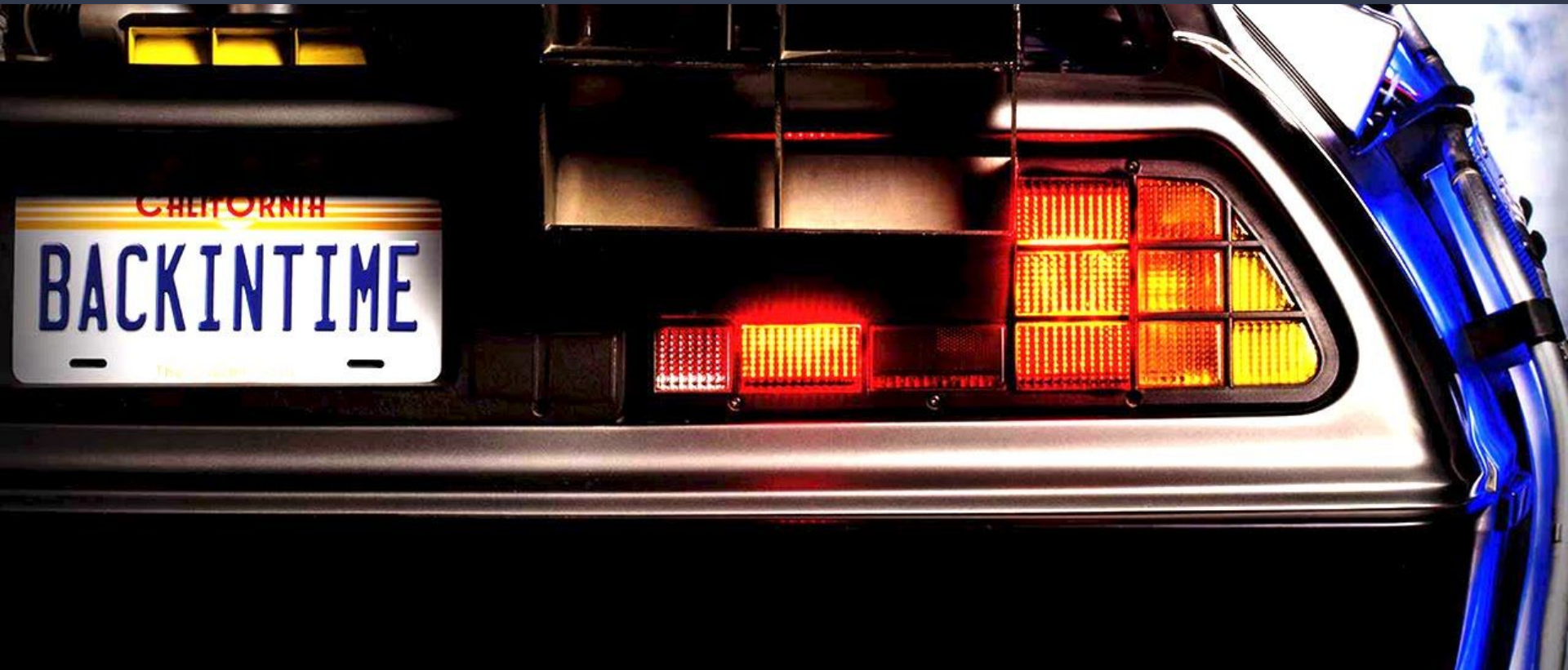
**Outsource to others, where possible**

THANK YOU!

Questions?

@barnhartguy

More after the break!



# Common Problems

Sharing Keys and Key Management

Splitting secrets

# Diffie Hellman Key Exchange

Alice and Bob want to create a shared secret  $s$ , but don't have anything to rely on

# Diffie Hellman Key Exchange

Agree on modulus and base ( $n=23$ ,  $g=5$ )

Alice: Prv{ $a=4$ }, send  $A=g^a \bmod n \Rightarrow A=5^4 \bmod 23=4$

Bob: Prv{ $b=3$ }, send  $B=g^b \bmod n \Rightarrow B=5^3 \bmod 23=10$

$$S_A = 10^4 \bmod 23 = 18$$

$$S_B = 4^3 \bmod 23 = 18$$

# Diffie Hellman Key Exchange

This is vulnerable to MitM

To better understand, look into STS variants

# Multi Party Computation

My password is DC9723



# Multi Party Computation

DC9\_2\_

DC\_7\_3

DC97\_3

DC9\_23

DC\_72\_

# Multi Party Computation

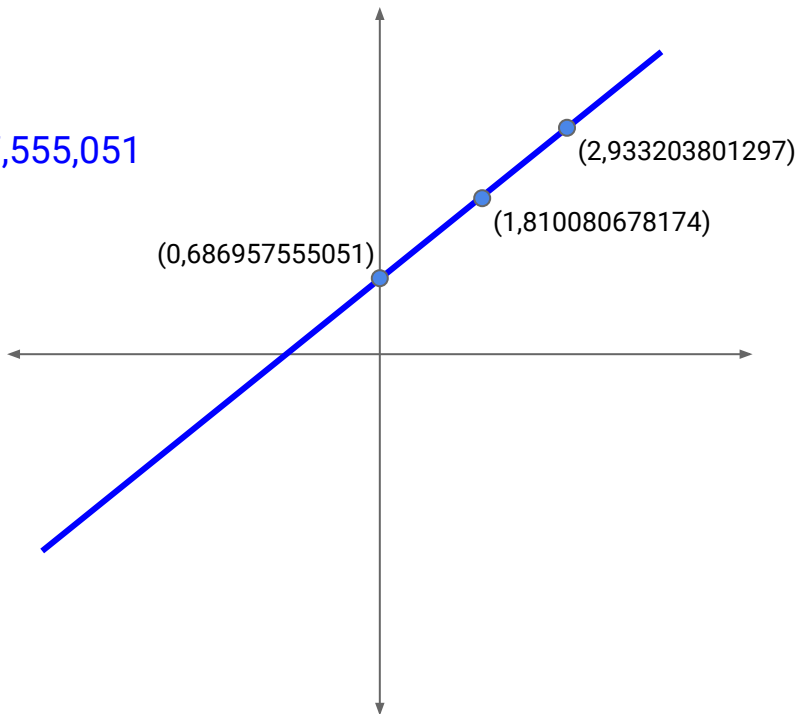
DC9723  $\Rightarrow$  68, 69, 57, 55, 50, 51  
686, 957, 555, 051

R = 123, 123, 123, 123 (gradient)

# Multi Party Computation (Shamir Secret Sharing)

$$y = ax + b$$

$$y = 123123123123 * x + 686,957,555,051$$



# Breaking Passwords

Passwords should not be stored in the clear

Hash(password)  $\Rightarrow$  can we still break?

KDF(password)  $\Rightarrow$  longer time to break?

KDF(password | salt | pepper)  $\Rightarrow$  even longer time to break?

# Hash Tables (Rainbow Tables)

Take all possible passwords,  
Build a database of {Hash(password), password} pairs

For any given hash, lookup in the database to find the  
password

Easy, right?

# Hash Tables (Rainbow Tables)

Compute time

Storage space

# Rainbow Table Attacks

`CipherText` = Encryption(`Message`, `Key`)

If I only have `CipherText`, can I break it?

If I know both the `Message` and the `CipherText`, can I break it?

# Rainbow Table Attacks



$M_1 = 00000000$

$C_1 = \text{Enc}(M_1, \text{Key})$



# CurveBall (CVE-2020-0601)



# ECC Certificate Validation

1. Choose a known good point  $G$  (generator/base point)
2.  $Q = d \times G \Rightarrow \text{Pub}\{Q, G\}, \text{Prv}\{d, G\}$
3.  $s = m \times d$  ( $m$  is the message to sign)
  - a.  $s \times G = (m \times d) \times G$
  - b.  $m \times Q$  can be computed
  - c.  $m \times Q$  should be equal to  $(m \times d) \times G$ 
    - i.  $(m \times d) \times G = m \times (d \times G) = m \times Q$

# ECC Certificate Validation

What if no-one verified that we use the same  $G$ ?

If you can choose any  $G$  you want, you can cheat!

$G' = Q/d'$  (remember  $Q = d \times G$ )

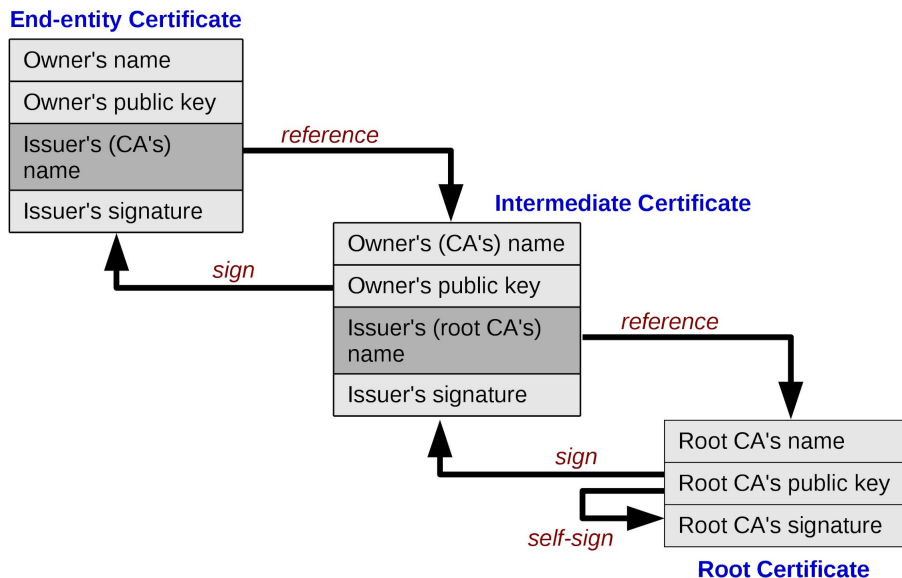
So...

$Q = d' \times G'$  and  $Q = d \times G$

If only  $Q$  is checked, we can sign using  $d'$

# ECC Certificate Validation

But a certificate with  $G'$  and  $Q$  is different than  $G$  and  $Q$ , right?



# ECC Certificate Validation

There are only two hard things  
in Computer Science: cache  
invalidation and naming things.

-- Phil Karlton

# ECC Certificate Validation

## Windows CryptoAPI Vulnerability

If the legitimate certificate is cached, only Q is compared for the certificate to be validated

# ECC Certificate Validation

No CVE if:

- only standard ECC parameters are allowed
  - No control over G
- Comparing all parameters (or digest)
  - No caching exploitation

# Thanks

Tal Be'ery - gread blog posts

Filo Sottile - Bitcoin vulnerability



THANK YOU!

Questions?

@barnhartguy