Digital Signatures

CMSC 23200/33250, Autumn 2018, Lecture 8

David Cash

University of Chicago

Plan

1. Digital Signatures Recall
2. Plain RSA Signatures and their many weaknesses
3. A Strengthing: PKCS#1 v1.5 RSA Signature Padding
4. An implementation error and its grave consequences

Assignment 1 is Due Tonight

Error in Problem 3 Hint:

- Technique outlined there omits an XOR with previous block.

If you want to test your code:

- Run attack with **`cnet_id=davidcash`** and **`cnet_id=ravenben`**
- Flag sizes vary in problems 2 and 3; Your attack should be robust to this
- (Especially on 2, where extra tricks are required for long flags.)

Crypto Tool: Digital Signatures

Definition. A digital signature scheme consists of three algorithms **`Kg`**, **`Sign`**, and **`Verify`**

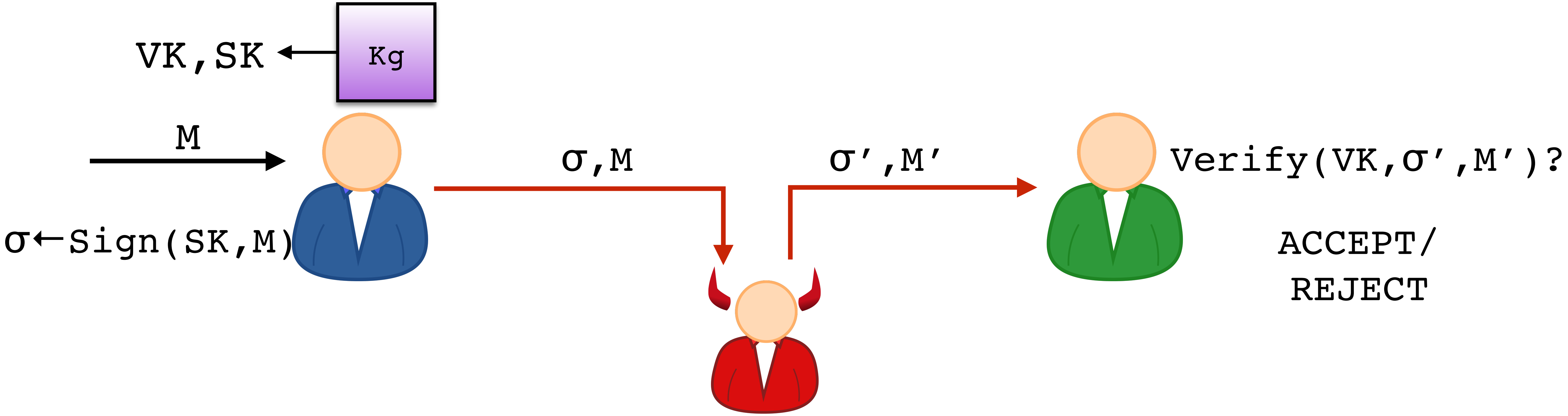
- Key generation algorithm **`Kg`**, takes no input and outputs a (random) public-verification-key/secret-signing key pair `(VK,SK)`

- Signing algorithm **`Sign`**, takes input the secret key `SK` and a message `M`, outputs "signature" $\sigma \leftarrow \texttt{Sign(SK,M)}$

- Verification algorithm **`Verify`**, takes input the public key V`K`, a message `M`, a signature σ, and outputs `ACCEPT/REJECT`

$$\texttt{Verify(VK,M,}\sigma\texttt{)=ACCEPT/REJECT}$$

Digital Signature Security Goal: Unforgeability



Scheme satisfies **unforgeability** if it is unfeasible for Adversary (who knows `VK`) to fool Bob into accepting `M′` not previously sent by Alice.

“Plain” RSA with No Encoding

☣ **Broken** ☣

$VK = (N, e)$ $\quad SK = (N, d)$ where $\quad N = pq, \quad ed = 1 \bmod \phi(N)$

$$\mathrm{Sign}((N, d), M) = M^d \bmod N$$

$$\mathrm{Verify}((N, e), M, \sigma) : \sigma^e = M \bmod N?$$

Messages & sigs are in $\mathbb{Z}_N^*$

$e = 3$ is common for fast verification; Assume `e=3` below.

“Plain” RSA Weaknesses

Broken

Assume $e=3$.

$$\mathrm{Sign}((N,d),M) = M^d \bmod N \qquad \mathrm{Verify}((N,3),M,\sigma) : \sigma^3 = M \bmod N?$$

To forge a signature on message M': Find number σ' such that $(\sigma')^3=M' \bmod N$

M=1 weakness: If $M'=1$ then it is easy to forge. Take $\sigma'=1$:

$$(\sigma'^3)=1^3=1=M' \bmod N$$ ✓

Cube-M weakness: If M' is a *perfect cube* then it is easy to forge. Just take $\sigma'=(M')^{1/3}$:, i.e. the usual cube root of M':

Example: To forge on $M'=8$, which is a perfect cube, set $\sigma'=2$.

$$(\sigma')^3=2^3=8=M' \bmod N$$ ✓

(Intuition: If cubing does not “wrap modulo N”, then it is easy to un-do.)

Further “Plain” RSA Weaknesses

Broken

$$\text{Sign}((N,d),M) = M^d \bmod N \qquad \text{Verify}((N,3),M,\sigma) : \sigma^3 = M \bmod N?$$

To forge a signature on message M': Find number σ' such that $(\sigma')^3 = M' \bmod N$

***Malleability* weakness:** If σ is a valid signature for M, then it is easy to forge a signature on $8M \bmod N$.

Given (M,σ), compute forgery (M',σ') as

$$M' = (8*M \bmod N), \text{ and } \sigma' = (2*\sigma \bmod N)$$

Then $\text{Verify}((N,3),M',\sigma')$ checks:

$$(\sigma')^3 = (2*\sigma \bmod N)^3 = (2^3*\sigma^3 \bmod N) = (2^3*M \bmod N) = 8M \bmod N \quad ✓$$

$\sigma^3 = M \bmod N$ b/c σ is valid sig. on M

Further “Plain” RSA Weaknesses

Broken

$\mathrm{Sign}((N,d),M) = M^d \bmod N$ $\mathrm{Verify}((N,3),M,\sigma) : \sigma^3 = M \bmod N?$

To forge a signature on message `M’`: Find number `σ’` such that $(\sigma')^3 = M' \bmod N$

***Malleability* weakness:** If σ is a valid signature for `M`, then it is easy to forge a signature on `8M mod N`.

General form of *malleability* weakness: If σ is a valid signature for `M`, then it is easy to forge a signature on `M’=(x*M mod N)` for any perfect cube `x`.

$$M' = x*M \bmod N, \text{ and } \sigma' = (x^{1/3}*\sigma \bmod N)$$

Then `Verify((N,3),M’,σ’)` checks:

$$(\sigma')^3 = (x^{1/3}*\sigma \bmod N)^3 = (x*\sigma^3 \bmod N) = (x*M \bmod N) = (M' \bmod N)$$ ✓

$\sigma^3 = M \bmod N$ b/c σ is valid sig. on `M`

Further "Plain" RSA Weaknesses

☣ **Broken** ☣

$$\mathrm{Sign}((N,d),M) = M^d \bmod N \qquad \mathrm{Verify}((N,3),M,\sigma) : \sigma^3 = M \bmod N?$$

<u>To forge a signature on message `M'`</u>: Find number `σ'` such that $(\sigma')^3 = M' \bmod N$

<u>Combining signatures</u> **<u>weakness:</u>** If σ_1 is a valid signature for M_1, and σ_2 is a valid signature for M_2…

… then it is easy to compute signature σ' on $M' = (M_1 * M_2 \bmod N)$

$M' = (M_1 * M_2 \bmod N)$ and $\sigma' = (\sigma_1 * \sigma_2 \bmod N)$

Then `Verify((N,3),M',σ')` checks:

$$(\sigma')^3 = (\sigma_1 * \sigma_2 \bmod N)^3 = (\sigma_1^3 * \sigma_2^3 \bmod N) = (M_1 * M_2 \bmod N) = (M' \bmod N)$$

b/c σ_1, σ_2 are valid sigs

✓

Further “Plain” RSA Weaknesses

Broken

$$\text{Sign}((N,d),M) = M^d \bmod N \qquad \text{Verify}((N,3),M,\sigma) : \sigma^3 = M \bmod N?$$

To forge a signature on message M': Find number σ' such that $(\sigma')^3 = M' \bmod N$

***Backwards signing* weakness:** Generate *some* valid signature by picking σ' first, and then defining $M' = (\sigma'^3 \bmod N)$

Then $\text{Verify}((N,3),M',\sigma')$ checks:

$$(\sigma')^3 = (M' \bmod N) \quad ✓$$

Further “Plain” RSA Weaknesses

☣ **Broken** ☣

$$\text{Sign}((N,d),M) = M^d \bmod N \qquad \text{Verify}((N,3),M,\sigma) : \sigma^3 = M \bmod N?$$

To forge a signature on message M': Find number σ' such that $(\sigma')^3 = M' \bmod N$

Summary:

- Plain RSA Signatures allow several types of forgeries
- It was sometimes argued that these forgeries aren’t important: If M is english text, then M' is unlikely to be meaningful for these attacks
- But often they are damaging anyway

RSA Signatures with Encoding

$VK = (N, e) \qquad SK = (N, d)$ where $N = pq, \quad ed = 1 \bmod \phi(N)$

$$\mathrm{Sign}((N,d),M) = \mathrm{encode}(M)^d \bmod N$$

$$\mathrm{Verify}((N,e),M,\sigma) : \sigma^e = \mathrm{encode}(M) \bmod N?$$

Messages & sigs are in $\mathbb{Z}_N^*$

encode maps bit strings to numbers in $\mathbb{Z}_N^*$

<u>Encoding needs to address:</u>

- Perfect cubes
- Malleability
- Backwards signing

Encoding must be chosen with extreme care.

☣ **Broken** ☣

RSA Signature Padding: PKCS #1 v1.5 (simplified)

Note: We already saw PKCS#1 v1.5 *encryption* padding. This is *signature* padding. It is different.

`N`: **n**-byte long integer.
`H`: Hash fcn with **m**-byte output. ← Ex: `SHA-256, m=32`

```
Sign((N,d),M):
1. digest←H(M) // m bytes long
2. pad←FF||FF||…||FF// n-m-3 'FF' bytes
3. X←00||01||pad||00||digest
4. Output σ = $X^d$ mod N
```

```
Verify((N,3),M,σ):
1. X←($σ^3$ mod N)
2. Parse X→aa||bb||Y||cc||digest
3. If aa≠00 or bb≠01 or cc≠00
     or Y≠$(FF)^{n-m-3}$ or digest≠H(M):
      Output REJECT
4. Else: Output ACCEPT
```

Encoding needs to address:

- Perfect cubes → The high-order bits + digest means X is large and random-looking, rarely a cube.
- Malleability → Stopped by hash, ex: `H(2*M)≠2*H(M)`
- Backwards signing → Stopped by hash: given `digest`, hard to find `M` such that `H(M)=digest`.

RSA Signature Padding: PKCS #1 v1.5 (simplified)

Note: We already saw PKCS#1 v1.5 *encryption* padding. This is *signature* padding. It is different.

`N`: `n`-byte long integer.
`H`: Hash fcn with `m`-byte output. ← Ex: `SHA-256, m=32`

```
Sign((N,d),M):
1. digest←H(M) // m bytes long
2. pad←FF||FF||…||FF// n-m-3 ‘FF’ bytes
3. X←00||01||pad||00||digest
4. Output σ = $X^d$ mod N
```

```
Verify((N,3),M,σ):
1. X←($σ^3$ mod N)
2. Parse X→aa||bb||Y||cc||digest
3. If aa≠00 or bb≠01 or cc≠00
       or Y≠$(FF)^{n-m-3}$ or digest≠H(M):
        Output REJECT
4. Else: Output ACCEPT
```

Introduces new weakness:

- Hash collision attacks: If `H(M) = H(M’)`, then …

`Sign((N,d),M) = Sign((N,d),M’)`

- i.e., can reuse a signature for `M` as a signature for `M’`

Now: A Buggy Implementation, with an Attack

- Padding check is often implemented incorrectly
- Enables forging of signatures on *arbitrary* messages

Real-world attacks against:

- OpenSSL (2006)
- Apple OSX (2006)
- Apache (2006)
- VMWare (2006)
- All the biggest Linux distros (2006)
- Firefox/Thunderbird (2013)

…

(too many to list)

Broken

Buggy Verification in PKCS #1 v1.5 RSA Signatures

```
Sign((N,d),M):
1. digest←H(M) // m bytes long
2. pad←FF||FF||…||FF// n-m-3 'FF' bytes
3. X←00||01||pad||00||digest
4. Output $\sigma = X^d \bmod N$
```

```
Verify((N,3),M,σ):
1. X←($\sigma^3$ mod N)
2. Parse X→aa||bb||Y||cc||digest
3. If aa≠00 or bb≠01 or cc≠00
       or Y≠$(FF)^{n-m-3}$ or digest≠H(M):
        Output REJECT
4. Else: Output ACCEPT
```

```
BuggyVerify((N,3),M,σ):
1. X←($\sigma^3$ mod N)
2. Parse X→aa||bb||rest
3. If aa≠00 or bb≠01:
        Output REJECT
4. Parse rest=$(FF)^p$||00||digest||…,
       where p is any number
5. If digest≠H(M): Output REJECT
6. Else: Output ACCEPT
```

Checks if **`rest`** starts with any number of **`FF`** bytes followed by a `00` byte.

If so, it takes the next **`m`** bytes as **`digest`**.

Correct: `X = 00 01 FF FF FF FF FF FF FF FF 00 <DIGEST>`

Buggy: `X = 00 01 FF 00 <DIGEST> <IGNORED ………… BYTES>`

One or more **`FF`** bytes

Attacking Buggy Verification

Broken

One or more `FF` bytes

Buggy: `X = 00 01 FF 00 <DIGEST> <IGNORED ………… BYTES>`

To forge a signature on message `M'`: Find number σ' such that

$$(\sigma')^3 = \texttt{00 01 FF 00 H(M') <JUNK> mod N}$$

We'll use one `FF` byte

`m` bytes long

`n-m-4` bytes free
for attacker to pick

Freedom to pick `<JUNK>` means we can take any σ' such that:

$$\texttt{00 01 FF 00 H(M') 00 ...... 00} \le (\sigma')^3 \le \texttt{00 01 FF 00 H(M') FF ...... FF}$$

Sufficient to find: Any perfect cube in the given range. Then apply perfect cube attack.

Easy! (exercise)

Steps in Attack

1. Pick `M` you want to forge on
2. Compute lower and upper bounds (numbers), using `H(M).`
3. Find a perfect cube `x` within allowed range
4. Output cube root of `x` as forged signature **σ**.

Attack Summary

- When padding check allows variable number of `FF` bytes, forging is easy
  - Only requires a simple search for a perfect cube in a given range
- *Why did so many make this error?*
  - I don't know
  - My guesses:
    - Plugging in libraries for padding removal without context
    - Traditional unit testing is hard to apply to crypto.
    - The details omitted in my description of the padding make parsing much harder. (Actual version includes in X `an` ASN.1 identifier of hash function, which is complicated in full generality.)
- Attack defeated by using large `e=65537`

Lesson with Implementing Signatures

- `Verify` should simply re-run signing and check if same signature comes out
- Not strictly possible if `Sign` is randomized.

Other RSA Padding Schemes: Full Domain Hash

N: **n**-byte long integer.
H: Hash fcn with **m**-byte output. ← Ex: `SHA-256, m=32`
`k = ceil((n-1)/m)`

```
Sign((N,d),M):
1. X←00||H(1||M)||H(2||M)||…||H(k||M)
2. Output σ = $X^d$ mod N
```

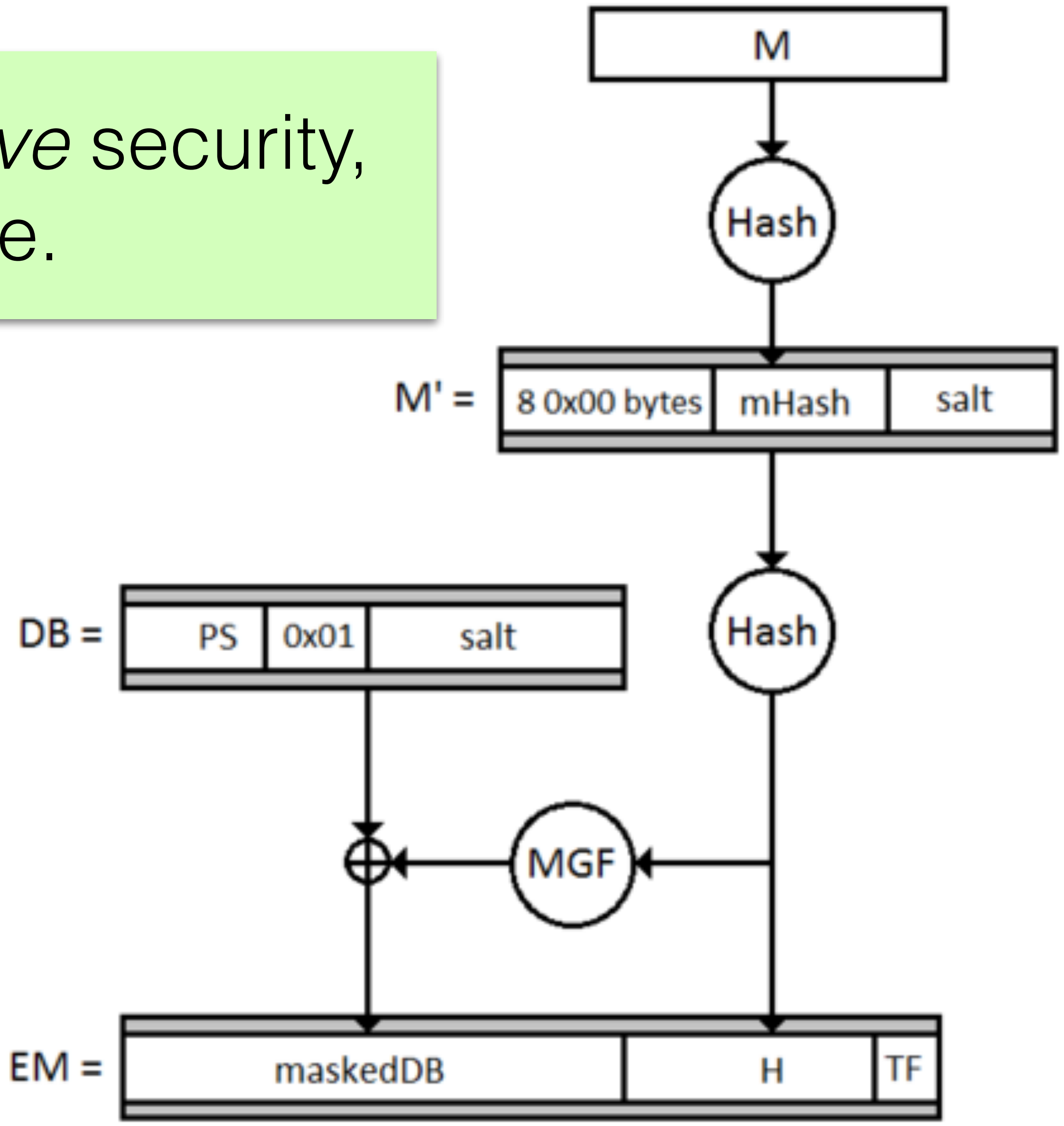
```
Verify((N,e),M,σ):
1. X←00||H(1||M)||H(2||M)||…||H(k||M)
2. Check if $σ^e$ = X mod N
```

Bonus: Can *prove* security, in a strong sense.

Other RSA Padding Schemes: PSS

- Somewhat complicated
- *Randomized* signing

Bonus: Can *prove* security, in a strong sense.

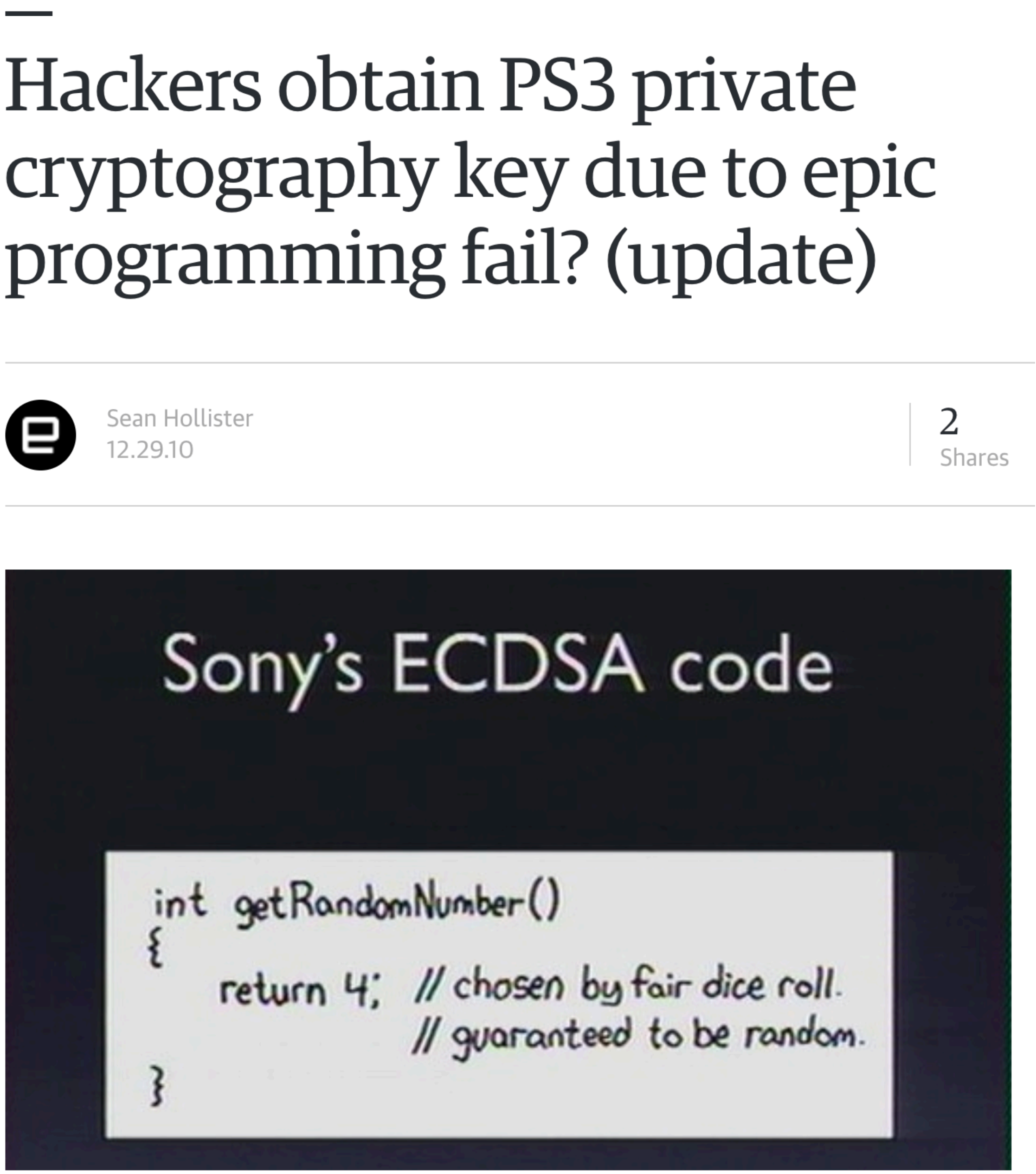


RSA Signature Summary

- Plain RSA signatures are very broken
- PKCS#1 v.1.5 is widely used, in TLS, and fine if implemented correctly
- Full-Domain Hash and PSS should be preferred
- Don’t roll your own RSA signatures!

Other Practical Signatures: DSA/ECDSA

- Based on ideas related to Diffie-Hellman key exchange
- Secure, but ripe for implementation errors



Hackers obtain PS3 private cryptography key due to epic programming fail? (update)

Sean Hollister
12.29.10

2
Shares

Sony's ECDSA code

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

The End