Recent (2014) vulnerabilities in SSL implementations

Introduction

- We will discuss two vulnerabilities in SSL implementations that were found in 2014:
  - The “Apple” bug, affecting recent Mac OS X and iOS devices.
  - The “heartbleed” bug concerning misuse of TLS protocol.
- First, some more background on SSL/TLS connections is discussed.

Server authentication

- In SSL/TLS, server authentication is performed using a server's certificate.
- The certificate contains among other things the hostname of the server and a signature.
- Client must check the hostname it connected to matches the hostname in the certificate.
- Client must check signature in the certificate.

Public Key Certificate

- These certificates are generated as follows:
  - A public/private key pair is generated for the server.
  - The public key is submitted to the CA as part of the CSR (certificate signing request).
  - The private key is kept private, not even the CA gets to see it.
  - CA generates a certificate containing the public key and a signature by the CA.

SSL/TLS handshake

- Recall the handshake used to establish secure connection between two hosts:
  - Server sends its certificate, random value, etc.
  - Client authenticates server
  - Client generates pre-master secret and encrypts this with server's public key (obtained from server certificate)
  - Server decrypts with its private key and the master (shared) secret is established

Problem

- Imagine the server's private key has been compromised by an attacker using any means **and** previous SSL/TLS communication has been recorded by this attacker.
- An attacker can now:
  - Decrypt the pre-master secrets sent by clients.
  - And thus derive the used master secrets.
- All previously recorded encrypted (“confidential”) communications can now be decrypted!

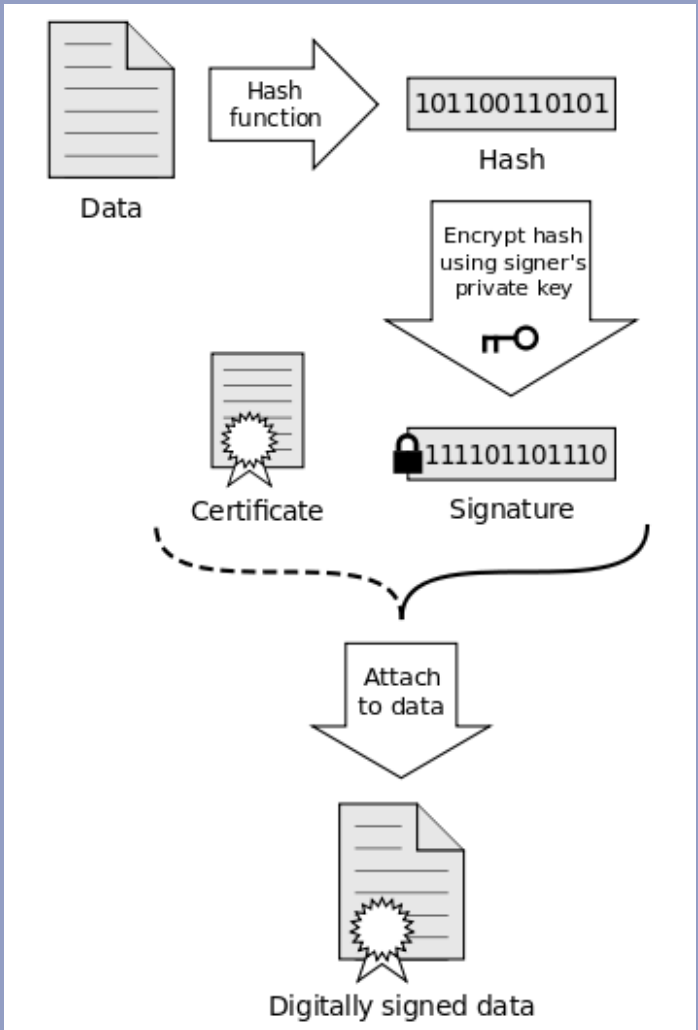
Cause and solution

- The server certificate is created only once based on just one private key.
- All SSL/TLS handshakes with different clients are carried out using the same certificate.
- Therefore, if the pre-master secret sent by these clients is also recorded, an attacker can decrypt all these keys with just this single private key.
- An attacker obtains all shared keys and can therefore decrypt all previous messages.
- Solution: use another mechanism to agree on a shared key.

Agreeing on shared secret

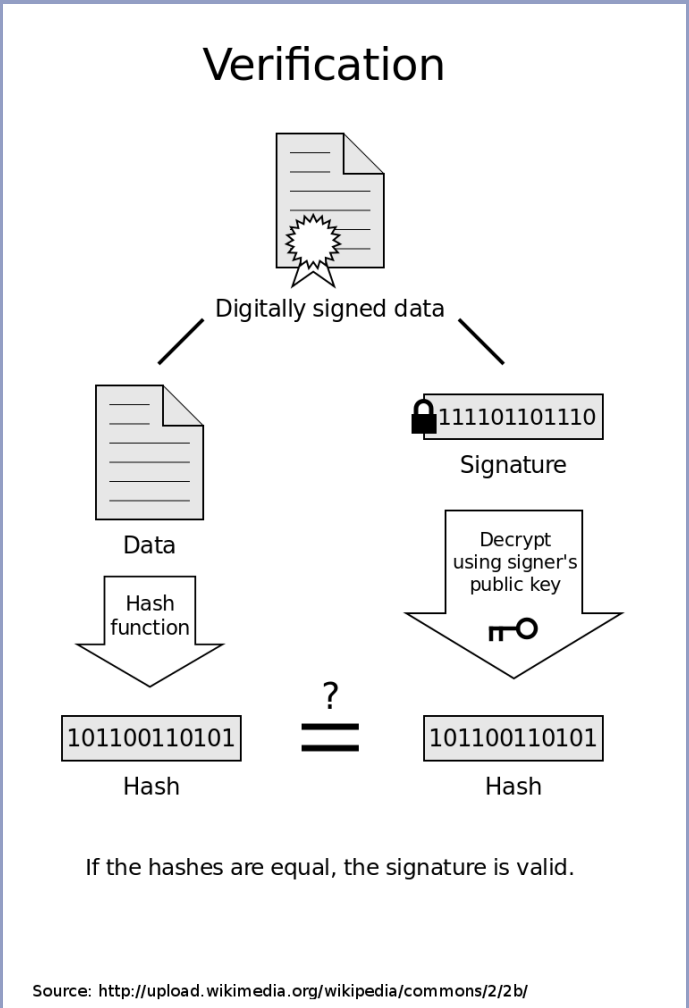
- We have discussed a protocol for this already: Diffie-Hellman key exchange.
- In TLS: server generates its secret 'a' as a random number, computes 'A' using 'a', and sends 'p', 'g' and 'A' values: ServerKeyExchange message.
- These values are sent in the clear, but signed with the server's private key.
- Client **must validate the signature** using the server's public key (from the certificate) to be certain 'p', 'g' and 'A' originate from this server.

Signing


Hash function
101100110101
Hash
Data
Encrypt hash using signer's private key
111101101110
Signature
Certificate
Attach to data
Digitally signed data

Verification of signed data

The hash function is initially agreed upon between server and client during SSL/TLS handshake, when selecting a cipher suite. An example is “SHA1”.



DH (Diffie-Hellman) Ephemeral

- The described Diffie-Hellman exchange always uses newly generated secrets “a” and “b”.
- This is referred to as Ephemeral Diffie-Hellman (DHE).
- For each connection an ephemeral (“temporary”) key is generated.

Forward Secrecy

- With DHE we can achieve forward secrecy:
  - if the private key of the server is comprised in the future, past communication remains secret.
- Why not always enable this? It is costly to achieve.
- However, Google and Twitter do have this enabled these days.

The “Apple” bug

- Apple has their own implementation of the SSL security protocol: “libsecurity”.
- Also referred to as “SecureTransport”.
- This is used on recent Mac OS X (10.9, used on MacBooks, etc.) and iOS (used on iPhones, etc.).
- In February 2014 a large vulnerability was found: the server signature for the shared secret was never validated.

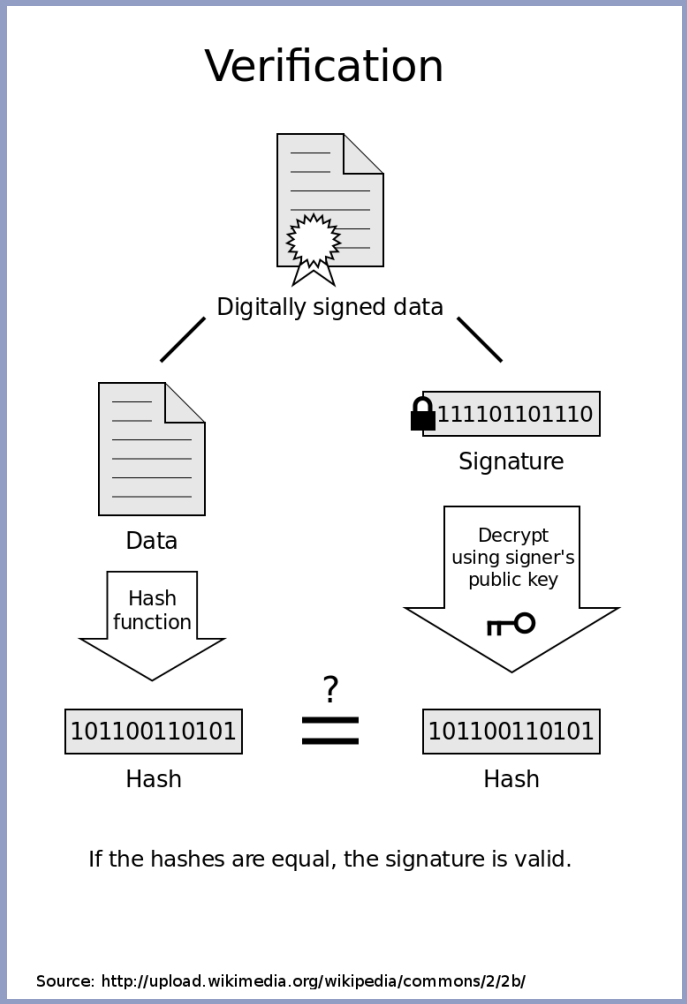
```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
                                 SSLBuffer signedParams,
                                 uint8_t *signature
                                 UInt16 signatureLen)
{
  OSStatus        err;
  ...

  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;
  ...
  /* signature verification */
  ...

fail:
  SSLFreeBuffer(&signedHashes);
  SSLFreeBuffer(&hashCtx);
  return err;
}
```

Verification of signed data

`SSLHashSHA1.final`



```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
                                 SSLBuffer signedParams,
                                 uint8_t *signature
                                 UInt16 signatureLen)
{
  OSStatus        err;
  ...

  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;
  ...
  /* signature verification */
  ...

fail:
  SSLFreeBuffer(&signedHashes);
  SSLFreeBuffer(&hashCtx);
  return err;
}
```

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
                                 SSLBuffer signedParams,
                                 uint8_t *signature
                                 UInt16 signatureLen)
{
  OSStatus        err;
  ...

  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;
  ...
  /* signature verification */
  ...

fail:
  SSLFreeBuffer(&signedHashes);
  SSLFreeBuffer(&hashCtx);
  return err;
}
```

- Third condition and further conditions are never checked!
- The signature is never verified!
- Goto fail, with return value indicating success.

Why is this a problem?

- Man-in-the-middle attacks on TLS connections are possible!
- Consider communication with your bank.
- An adversary may send your bank's certificate, but its own signed (or even unsigned!) secret.
- The “lock” in your browser window is shown, even though you might not be **DIRECTLY** communicating with your bank's server.

XKCD Cartoon (xkcd.com)

- Some programmers blame this on the use of goto (although the goto use in this case can be seen as appropriate).
- Other programmers re-iterate that you should always use curly braces in if-statements in C.

The “heartbleed bug”

- Discovered and publicized in April 2014.
- Problem is in the “heartbeat” extension of TLS (RFC 6520).
- Implementation in OpenSSL is buggy: allows memory of system to be read.
- Linux and BSD systems rely on OpenSSL, so this bug is VERY widespread.

TLS heartbeat

- Send a packet requesting a response that echoes the payload of the packet.
- Are you alive? If so, send back these *n* bytes from my packet.

Why was this introduced?

- From RFC 6520:

> The only mechanism available at the DTLS layer to figure out if a peer is still alive is a costly renegotiation, particularly when the application uses unidirectional traffic. Furthermore, DTLS needs to perform path MTU (PMTU) discovery but has no specific message type to realize it without affecting the transfer of user messages.

DTLS is TLS for Datagrams (e.g. UDP)

> TLS is based on reliable protocols, but there is not necessarily a feature available to keep the connection alive without continuous data transfer.

> **Even in the case of TLS/TCP, this allows a check at a much higher rate than the TCP keep-alive feature would allow.**

Why was this introduced?

- Alternatively: you could always implement your own keep-alive support in the application layer, without needing to touch the SSL/TLS layer.

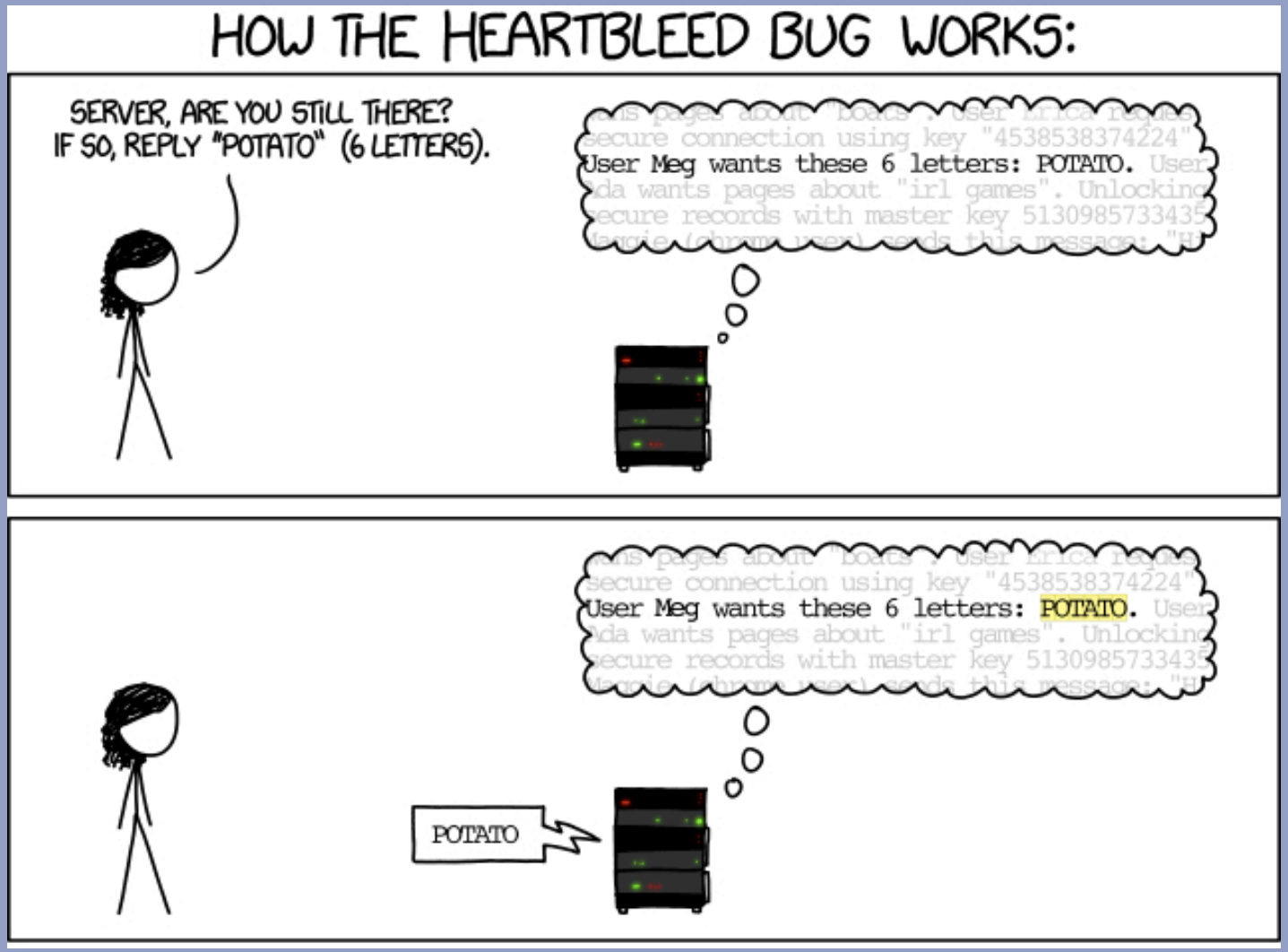
TLS heartbeat packet format

- The heartbeat protocol is defined on top of the TLS "Record" layer.
  - ContentType: 1 byte
  - Protocol version: 2 bytes
  - Length: 2 bytes
  - Protocol message: *"length"* bytes

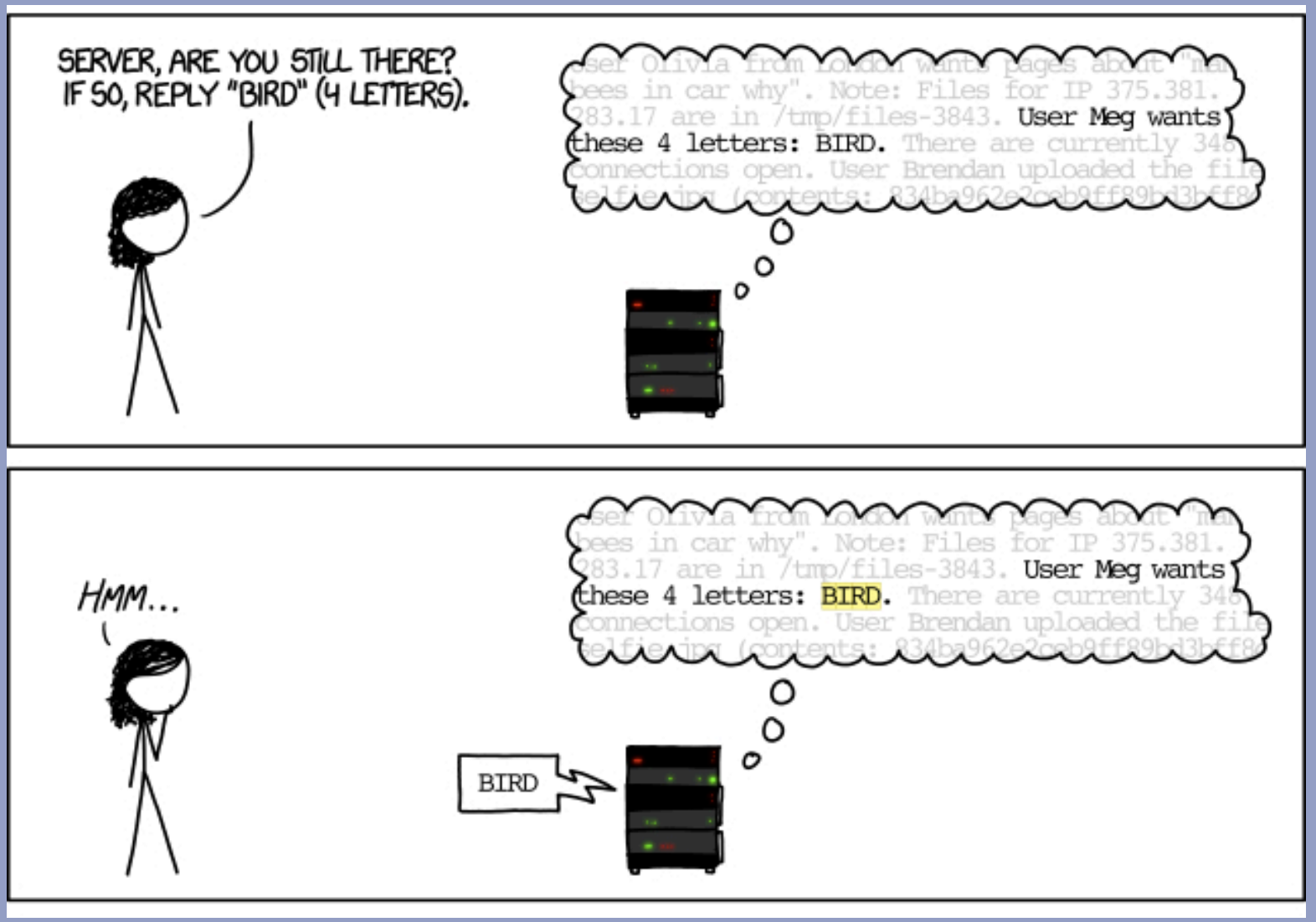
TLS heartbeat packet format

- The heartbeat protocol is defined on top of the TLS "Record" layer.
  - ContentType: 1 byte
  - Protocol version: 2 bytes
  - Length: 2 bytes
  - Protocol message: *"length"* bytes
    - HeartbeatMessageType: 1 byte
    - Payload length: 2 bytes
    - Payload: *"payload_length"* bytes

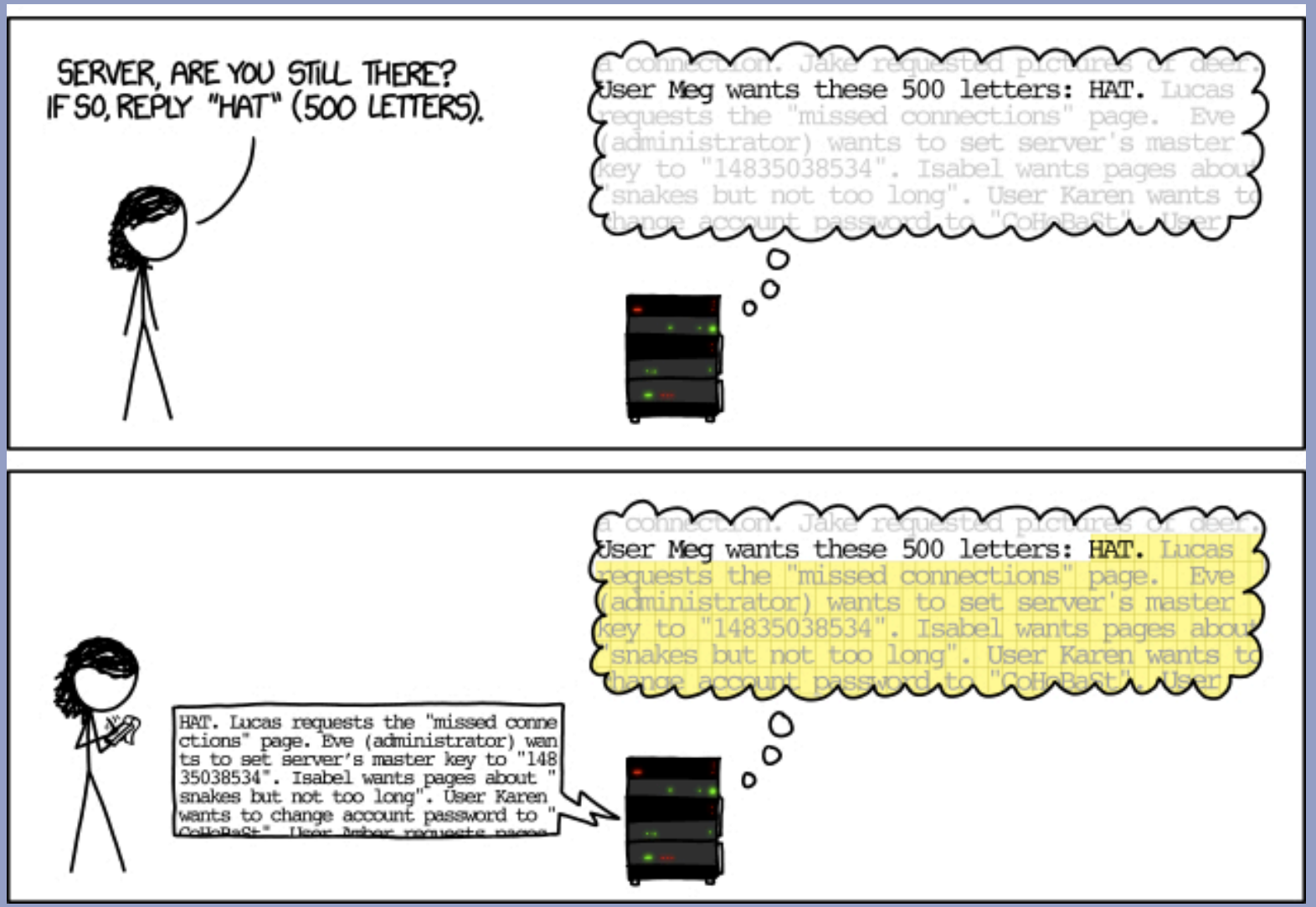
Cartoon


HOW THE HEARTBLEED BUG WORKS:
SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).
User Meg wants these 6 letters: POTATO.
POTATO

Cartoon


SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
User Meg wants these 4 letters: BIRD.
HMM...
User Meg wants these 4 letters: BIRD.
BIRD

Cartoon


SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
User Meg wants these 500 letters: HAT.
HAT. Lucas requests the "missed conne ctions" page. Eve (administrator) wan ts to set server's master key to "148 35038534". Isabel wants pages about " snakes but not too long". User Karen wants to change account password to "

TLS heartbeat packet format

- The heartbeat protocol is defined on top of the TLS "Record" layer.
  - ContentType: 1 byte
  - Protocol version: 2 bytes
  - **Length: 2 bytes**
  - Protocol message: *"length"* bytes
    - HeartbeatMessageType: 1 byte
    - **Payload length: 2 bytes**
    - Payload: *"payload_length"* bytes

The problem

- Send a heartbeat packet with a payload of *n* bytes, claiming the payload is *m* bytes with *n* < *m*.
- So “length” < “payload_length”.

Code

At this point, the protocol message (heartbeat message) is parsed:

```
/* Read type and payload length first */
hbtype = *p++;
n2s(p, payload);
pl = p;

...

buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;

*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
bp += payload;
```

Read 2 bytes from “p” and write into “payload” variable.

Code

```
/* Read type and payload length first */
hbtype = *p++;
n2s(p, payload);
pl = p;

...

buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;
```

Here, a reply to the heartbeat message is formed:

```
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
bp += payload;
```

Copy “payload” bytes from p1 to bp

The payload value received in the packet is never checked!

The bug

- OpenSSL:
  - Did check the “length” field.
  - Did not check the “payload_length” field.
- This allows packets with n < m to be processed!
- This triggered a memory read overrun!

Fixed Code

(s->s3->rrec.length is the length field of the TLS packet).

```
/* Read type and payload length first */
if (1 + 2 + 16 > s->s3->rrec.length)
  return 0; /* silently discard */
hbtype = *p++;
n2s(p, payload);
if (1 + 2 + payload + 16 > s->s3->rrec.length)
  return 0; /* silently discard per RFC 6520 sec. 4 */
pl = p;
```

References

- Cartoons: http://www.xkcd.com/
- RFC 5246
- https://www.imperialviolet.org/2014/02/22/applebug.html
- https://polarssl.org/kb/cryptography/ephemeral_diffie_hellman
- http://vincent.bernat.im/en/blog/2011-ssl-perfect-forward-secrecy.html
- https://gotofail.com/faq.html

- RFC 6520
- http://git.openssl.org/gitweb/?p=openssl.git;a=commitdiff;h=96db902
- http://www.theregister.co.uk/2014/04/09/heartbleed_explained/