Security 14: TLS, OTP

i.g.batten@bham.ac.uk

Recap

- Attackers can intercept packets
- Attackers can modify packets
- Attackers can inject packets
- Attackers can discard packets
- “A Dolev Yao attacker”

TLS

- “Transport Layer Security”
- Standardisation and extension of Netscape’s “SSL” Secure Sockets Layer.
- Idea is to provide an encrypted layer over TCP, so that applications that run over TCP will easily run over TLS and get all of TCP’s properties and CIA.
- Lots of security implications and, as ever, crypto is hard to do right.

TLS has lots of options

- Basic idea is that we use:
  - asymmetric encryption to agree a key,
  - some sort of zero knowledge proof to show possession of private key for certificate (might be combined with previous stage)
  - symmetric encryption to encrypt data
    - with HMAC or hash to prove integrity
- Some combinations are weak, and implementations have flaws
- TLS supports huge range of negotiated cipher suites.

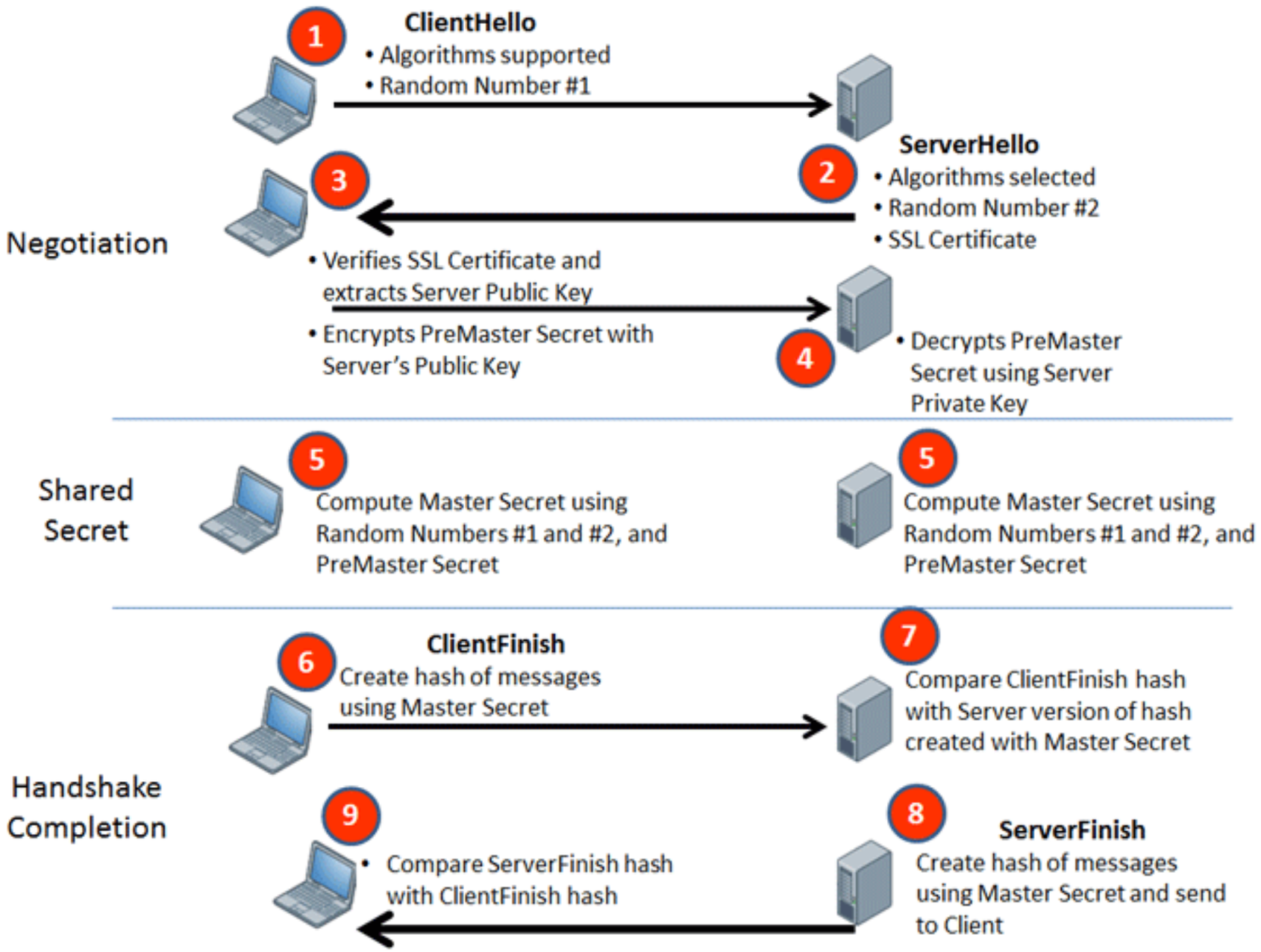
Want to bet on all of these?

```
ians-macbook-air:NETSEC16 igb$ openssl ciphers | tr ':' '
' | sort | pr -t -4 -w 132 | col -x
AES128-GCM-SHA256               DH-RSA-CAMELLIA128-SHA          ECDH-ECDSA-AES256-GCM-SHA384    ECDHE-RSA-AES256-SHA
AES128-SHA                      DH-RSA-CAMELLIA256-SHA          ECDH-ECDSA-AES256-SHA           ECDHE-RSA-AES256-SHA384
AES128-SHA256                   DH-RSA-DES-CBC3-SHA             ECDH-ECDSA-AES256-SHA384        ECDHE-RSA-DES-CBC3-SHA
AES256-GCM-SHA384               DH-RSA-SEED-SHA                 ECDH-ECDSA-DES-CBC3-SHA         ECDHE-RSA-RC4-SHA
AES256-SHA                      DHE-DSS-AES128-GCM-SHA256       ECDH-ECDSA-RC4-SHA              EDH-DSS-DES-CBC3-SHA
AES256-SHA256                   DHE-DSS-AES128-SHA              ECDH-RSA-AES128-GCM-SHA256      EDH-RSA-DES-CBC3-SHA
CAMELLIA128-SHA                 DHE-DSS-AES128-SHA256           ECDH-RSA-AES128-SHA             IDEA-CBC-SHA
CAMELLIA256-SHA                 DHE-DSS-AES256-GCM-SHA384       ECDH-RSA-AES128-SHA256          PSK-3DES-EDE-CBC-SHA
DES-CBC3-SHA                    DHE-DSS-AES256-SHA              ECDH-RSA-AES256-GCM-SHA384      PSK-AES128-CBC-SHA
DH-DSS-AES128-GCM-SHA256        DHE-DSS-AES256-SHA256           ECDH-RSA-AES256-SHA             PSK-AES256-CBC-SHA
DH-DSS-AES128-SHA               DHE-DSS-CAMELLIA128-SHA         ECDH-RSA-AES256-SHA384          PSK-RC4-SHA
DH-DSS-AES128-SHA256            DHE-DSS-CAMELLIA256-SHA         ECDH-RSA-DES-CBC3-SHA           RC4-MD5
DH-DSS-AES256-GCM-SHA384        DHE-DSS-SEED-SHA                ECDH-RSA-RC4-SHA                RC4-SHA
DH-DSS-AES256-SHA               DHE-RSA-AES128-GCM-SHA256       ECDHE-ECDSA-AES128-GCM-SHA256   SEED-SHA
DH-DSS-AES256-SHA256            DHE-RSA-AES128-SHA              ECDHE-ECDSA-AES128-SHA          SRP-3DES-EDE-CBC-SHA
DH-DSS-CAMELLIA128-SHA          DHE-RSA-AES128-SHA256           ECDHE-ECDSA-AES128-SHA256       SRP-AES-128-CBC-SHA
DH-DSS-CAMELLIA256-SHA          DHE-RSA-AES256-GCM-SHA384       ECDHE-ECDSA-AES256-GCM-SHA384   SRP-AES-256-CBC-SHA
DH-DSS-DES-CBC3-SHA             DHE-RSA-AES256-SHA              ECDHE-ECDSA-AES256-SHA          SRP-DSS-3DES-EDE-CBC-SHA
DH-DSS-SEED-SHA                 DHE-RSA-AES256-SHA256           ECDHE-ECDSA-AES256-SHA384       SRP-DSS-AES-128-CBC-SHA
DH-RSA-AES128-GCM-SHA256        DHE-RSA-CAMELLIA128-SHA         ECDHE-ECDSA-DES-CBC3-SHA        SRP-DSS-AES-256-CBC-SHA
DH-RSA-AES128-SHA               DHE-RSA-CAMELLIA256-SHA         ECDHE-ECDSA-RC4-SHA             SRP-RSA-3DES-EDE-CBC-SHA
DH-RSA-AES128-SHA256            DHE-RSA-SEED-SHA                ECDHE-RSA-AES128-GCM-SHA256     SRP-RSA-AES-128-CBC-SHA
DH-RSA-AES256-GCM-SHA384        ECDH-ECDSA-AES128-GCM-SHA256    ECDHE-RSA-AES128-SHA            SRP-RSA-AES-256-CBC-SHA
DH-RSA-AES256-SHA               ECDH-ECDSA-AES128-SHA           ECDHE-RSA-AES128-SHA256
DH-RSA-AES256-SHA256            ECDH-ECDSA-AES128-SHA256        ECDHE-RSA-AES256-GCM-SHA384
ians-macbook-air:NETSEC16 igb$
```

Certificates

- A public key, signed by a certification authority
- I send you a certificate
- You check the signature (compare hash of certificate with signed value, using known CA key)
- You send me a random, encrypted with my public key
- I decode the random and do something to prove knowledge of it (send back the random in clear, use it to encrypt a known string, etc, etc)

Sample handshake



RSA key establishment

- Client generates a random number
- Encrypts it with server's public key
- Sends result to server
- Both parties now share the random and can generate a session key from it
- Also proves server has private key matching public key

Problem #1

- Clients are rubbish at generating random numbers
  - Without hardware RNG, best source of randomness is turbulence in disk drives
  - So useless for portable devices
- Sadly, not fixable with current protocols, but feeds into the more serious problem…

Problem #2

- Communication starts with session key (or material used to make session key) encrypted with server public key
- Server public key (therefore also private key) is used long-term (potentially years) because certificates have long lifetimes
- Therefore **later** compromise (including warranted) of private key reveals session keys for all previous sessions, if they have been intercepted
- Problem is referred to as **forward secrecy**

Partial Solution

- Diffie-Hellman key exchange
- Pre-agree prime p and generator g
  - Each party generates a random
  - Sends g^x mod p, g^y mod p
  - Computes (g^x)^y mod p and (g^y)^x mod p
- Both parties now share g^(xy) mod p

Diffie-Hellman

- Usually done with elliptic curves, not integers mod p.
- Only as strong as the weakest random number generator, sadly
- Doesn’t provide any proof of knowledge of long-term keys
  - So RSA is used with the certificate to prove key ownership

Lots of TLS Attacks

- Lucky 13
- BEAST
- POODLE
- Heartbleed
- Some attacks on crypto, some on implementation
- Complex protocol plus complex implementation plus complex crypto = lots of bugs
- “Downgrade attacks” also difficult to deal with: convince client that server only offers weak crypto, or vice versa.

Main implementation is a mess

- Most popular implementation is OpenSSL: has portability hacks for ancient, long-dead operating systems.
- Read up on LibreSSL and “Open SSL Rampage” for more details
- Read up on Debian SSL Random Number Generator bug CVE-2008-0166 to see myth of open source

Waterloo TCP supported in OpenSSL, last touched 15 years ago, TCP/IP for DOS and Windows 3.1

This page contains my port of Waterloo tcp/ip (WatTCP). Watt-32 is an enhanced version of Geof Cooper's TinyTCP and Erick Engelke's WatTCP. The latest version is dated November 1999 with features integrated into Watt-32.
Watt-32 is a library for making networked TCP/IP programs in the language of C and C++ under DOS and Windows-NT. Both 16-bit real-mode and 32-bit protected-mode is supported.
For DOS, Watt-32 requires a packet-driver (*PKTDRVR*) to access the data-link layer (Ether-PPP, SLIP or Ethernet. Token-Ring is un-tested). With the correct packet-driver, it will run under *all* versions of Windows too. I highly recommend SwsVpkt which works much faster than Dan Lanciani's NDIS3PKT. With the SwsVpkt or NDIS3PKT drivers one can connect to Windows services (on the same machine) from a DOS-box too.
For Windows, WinPcap and *NPF.SYS* are required. Note that Windows 95, 98 and ME are not supported.
The name Watt-32 was chosen to signal the emphasis on 32-bit platforms (Although 16-bit compiler are also supported). What, besides embedded systems, is DOS good for these days if not running high-performance 32-bit programs. And the embedded market is booming; with the price of PC-104/Ethernet cards, Watt-32 could be used in a lot of fancy boxes. How about an *IP-telephone*, *MP3 home-player* or an *Internet Radio*?

Alternatives

- GnuTLS: has history of bugs almost as bad as OpenSSL
- PolarSSL: used in embedded systems, less bad history, but much less well tested
- LibreSSL looking better and better twelve months in, so probably now the go-to implementation

HSMs

- Biggest problem is ensuring security of private key associated with certificate for domain
- Hardware Security Modules can store a key and perform only the appropriate tasks with it (use it to sign / decrypt but not release it)
- Could use a TPM but there are practical problems
- Unfortunately, CAs use HSMs, few other people do: more commonly a file in /etc…

TLS Benefits

- Protects against passive attackers observing valuable traffic (TLS offers **confidentiality**)
- Protects against active attackers modifying valuable traffic (TLS offers **integrity**)
- Protects against man-in-the-middle attacks if certificates are properly checked (if, **if**, **IF**).

TLS Problems

- Avoids all content filtering and checking services
  - Solutions to this worse than the problem
- Exceptionally reliant on correct issuing of certificates
- Typical “geek” security measure: provides good properties when implemented by experts and used by trained people, but full of pitfalls for “ordinary” developers and particularly for untrained users
  - “Write a guide for your grandmother to allow her to accurately check that a site has a correct certificate” is a good exam question, but not one I’d want to mark.
- This isn’t a security usability course, but TLS is one of the worst areas for usable security.

Certificate Issuance

- Certificate Transparency: all CAs log the certificates they issue into a public log, so that you can
  - Check that certificates were properly issued
  - Look for people mis-issuing “your” certificate
- Certificate Authority Authorisation: you can declare in your DNS who is allowed to issue certificates for you, and all CAs must check
  - However, you can simply not do this, which means “anyone can issue”, and so far no-one (fsvo no-one) is doing it.
  - And most people don’t do DNSsec

TLS Usage

- Can either start up immediately, and then have application run over the top (ie, connect to port 993, negotiate TLS session, then start IMAP
- Or connect to port 143, talk unencrypted IMAP, then say STARTTLS and switch to encrypted communications, over the same TCP connection, still over port 143.
- Nicer for firewalls, as can see early stages, and one port better than two.

Authentication over Networks

- What are we using TLS to protect?
- Often, just login details
  - For most people, it is much more important that people cannot create messages from you then confidentiality of messages to you
  - I'd rather people didn't read (or indeed modify) my bank statements, but the harm is small; I really don't want people spending money as me.
    - Yes, exotic attacks possible involving consistently changing bank statements to conceal other fraud: much more effective with malware, as relies on seeing all traffic for all sessions.

TLS Authentication

- Client-side certificates are rarely used, which is a shame (I have only ever used one once, for a CA which – ironically – is now on Mozilla's final warning list)
- Stored protected either by OS facilities by a static passphrase
- Passphrases can be obtained with cameras, key-loggers, malware, etc, but obtaining the private key requires malware (probably main attack vector)
- Appeal is that the only material stored on server is a public key, whose theft is less serious that (say) a password hash.
- Enrolment is hard (per-app? per-server? per-user, but do I trust other issuers? What issuers?)
- Sadly, adoption is non-existent

One Time Passwords

- Basic principle:
  - Secure piece of hardware (“token”) shares a secret with a server
  - Token combines (hash, encryption) secret with counter or clock, and displays the result
  - Used as password: server can check as it knows counter/clock and secret (crucially: it **knows the secret**, not a hash of the secret)

Implementation Details

- Button might have been pushed off-line, or clocks might have drifted
- Accept the expected password plus the next *n*, and update on server the expected offset
- Can add PIN feature on token to either unlock it, or combine PIN into hash as well.

Benefit

- Token is never connected to computer (or only connected “one-way” —Yubikeys pretend to be a keyboard)
- Secret cannot (fsvo cannot) be extracted, so token cannot be copied
- So long as server is secure, requires possession of token to log in

Problems

- Expensive (even $10 per token is expensive in large deployment) although soft-tokens help if security acceptable
  - Soft tokens rely heavily on security of app data on phones, good in iPhone >= 5S, somewhat random on Android.
- Server full of plaintext secrets is a major target
- Difficult to share between domains, so you end up with a fistful of tokens (worn around neck by geeks)
- Not hard to use, but not easy to use either

One Time Over the Air

- You attempt to log in as “igb”
- System texts password to phone number registered to “igb”
- igb logs in using one-time password
- Requires possession of phone (or, more precisely, SIM private key — see recent news stories about GCHQ attacks on Gemauto) or control of GSM network.

Benefits

- Phones are essentially universal amongst people likely to be using such services
- Solves problem of needing lots of tokens
- SIM security is well proven (GCHQ attacks on card issuers in a way proves how good the security is, as if security weak they could break it anyway)

Problems

- SMS messages require infrastructure in the data centre
- Roaming costs
- Delay in transmission can be “just bad enough” to make usability poor (especially if roaming)
- SS7 and other parts of GSM infrastructure a cesspit: CESG/NSCS advice is to not do this, and certainly not for anything protectively marked.

Popular for…

- Either as text or as voice call speaking the number, popular for authentication of new instructions to bank or (I found yesterday) setting up Apple Pay.
- Probably good enough for this purpose, but not for anything more serious

Protects Against

- Key loggers (different password each time)
  - Hence cameras, malware, etc
- Disclosure to others, sort of…
  - Requires physical possession of tag
  - Not resistant to shared use by phone

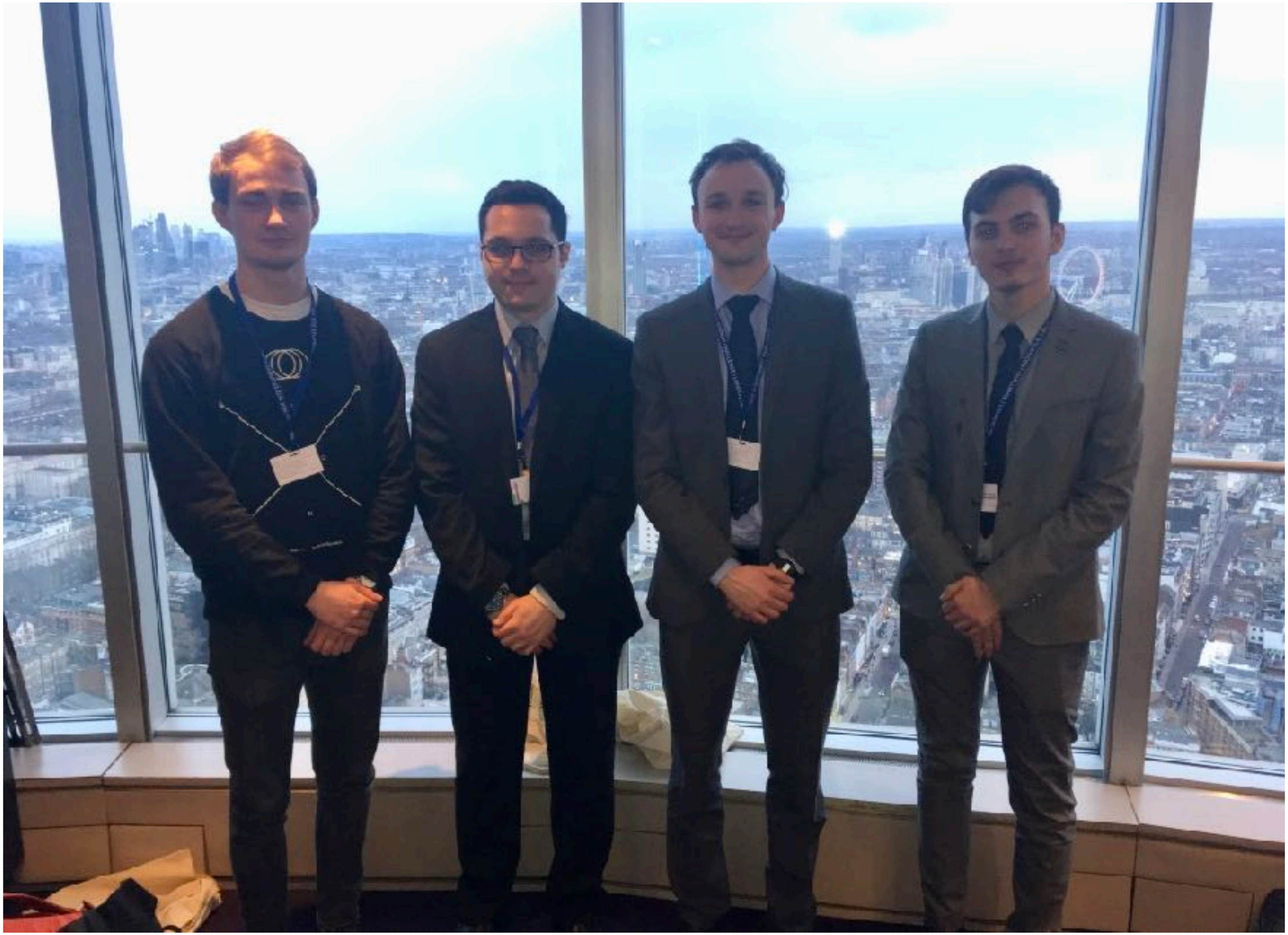
Breaks

- Original product “SecureID” used DES in some custom mode: vague suspicions might be attackable
- Now there are standards for Time and Counter based security, in “Google Authenticator” and similar.
- Some tags are “Time Bombed”: tags die after 3 years, whether battery is OK or not
    - Probably a good policy, although expensive for users

Phishing Works

- Various “seeds” (the initial secret) have been stolen, and users have to inherently trust the manufacturer
- SecureID normally used with a PIN simply concatenated with the code: provides little additional security
- Yubikey can be reprogrammed with new secret material, which helps


VASCO
880373
DEBIT/CREDIT CARD SALES VOUCHER
M 76851
T ****5077
Qty
001
Number
Total
£4.10C
Date
11-OCT-17
39085
Customer
VISA
Printed 08:13 on 11-OCT-17



Weaker form

- CRAM-MD5, DIGEST-MD5
- Server knows user password **IN PLAIN TEXT**
- Server sends random
- Client hashes random with password and replies
- Server repeats computation and checks knowledge of password

CRAM-MD5

- Not as good as a token: single, unchanging password
- Secure against an observer without needing complex crypto
- Requires unhashed secrets on server (but so do hardware tokens)
- Popular for IMAP and SMTP as good fit to protocol
  - Make sure email password is not your main password if CRAM-MD5 or DIGEST-MD5 are in use

Another weaker form

- ssh public key encryption
- ssh user holds private key
- ssh server holds public key
- server sends random, which is signed by client, checked by server (roughly: it's actually more complex than that)
  - Can use "signature only" algorithms like DSS/DSA

Problems

- Private key held by client on machine of unknown security
- Key is encrypted with passphrase chosen by user, typed by user on their own computer
  - And often stored in an “agent” unencrypted
- No publicly known breaks, but probably not the best way to secure logins from home/BYOD machines to machines holding high-value data.

Password Tunnelling

- Sending plaintext passwords over an encrypted channel feels bad
  - Prone to client-side malware
- But may not be as bad as it looks
- Risk analysis, risk analysis, risk analysis

OAUTH

- Mechanism for doing federated authentication: “I will accept you are MrX@SiteY, because Site Y says so”.
- Technology behind “Log in with Google / Twitter / Facebook”.
  - And leverages their two-factor authentication.
- Focussed on HTTP authentication but can be used for other things
  - Current project to add it to ssh, for example

OAUTH

- Complex to implement, hard to implement right
- Basic protocol looks sound (I modelled it in ProVerif, although I haven't published it)
- Lots of places to get implementations wrong, and my gut feel is that there are likely to be a lot of "practical" attacks at the interfaces between the components and in the (relatively complex) crypto: ProVerif doesn't model this.
- But currently looks like a good option to examine
- https://oauth.net

Conclusion

- TLS complex, bug-ridden, but only serious contender if you need confidentiality and integrity
- One time passwords with tokens may be enough for some applications: risk assessment required
- Does no harm to use one time passwords over TLS!
- Two factor authentication: always good