Towards safer security APIs

Chris Mitchell
Information Security Group
Royal Holloway, University of London

Agenda

1. Introduction
2. Security APIs – what is the problem?
3. New problems
4. Encryption and integrity – a case study
5. Rules for designing better APIs
6. Concluding remarks

Basic (informal) model

- 'Untrusted code' has access to a security API (i.e. a set of commands), which is supported using a set of secrets.
- Goal is to design the set of commands so that untrusted code cannot access the secrets themselves, or perform any unintended actions using the secrets.
- Security APIs are sometimes referred to as crypto APIs.
- Clulow (2003) provides an excellent discussion of basic models for security APIs.

Possible additional requirements

- Secure (integrity and confidentiality protected) storage of data and keys in untrusted environment.
- Key management functions (multilevel API and/or separate key management API?).
- Secure auditing (externally stored log).
- Enumeration of failed commands (e.g. indicative of exhaustion attacks on weak passwords).

State issues

- General principle: eliminate (minimise) need for command-related state.
- Enables support of API using multiple security modules with access to same set of secrets.
- This requirement conflicts (to some extent at least) with desire to count failed commands.

Agenda

1. Introduction
2. Security APIs – what is the problem?
3. New problems
4. Encryption and integrity – a case study
5. Rules for designing better APIs
6. Concluding remarks

Poorly designed APIs

- Designing robust security APIs appears to be a non-trivial problem.
- Many examples of attacks are known on poorly designed security APIs.
- Certainly need to find ways of addressing this problem.
- But what is the real problem?

Established wisdom

- It is often suggested that problems arise because of unintended interactions between commands.
- Certainly this is a potential issue, and addressing this is a tough problem.
- However, is this where the known problems lie?
- To check this, we briefly look at some known attacks on security APIs …

A brief survey of flaws I

- PIN cracking attacks, e.g.
  - Bond-Zielinski (2002);
  - Clulow (2003);
  - Berkman-Ostrovsky (2007);

  such attacks seems to use only a very small set of PIN-specific commands.
- However, should note that two of the PIN formats were individually secure, but when an HSM supports both (and enables translation between them) attacks become possible.

Survey of flaws II

- Key management API problems, e.g.:
  - Bond (2001), Clayton-Bond (2002) and Clulow (2003) – range of key management attacks;
  - Bond-Anderson (2001) – attack on key-share use;

these attacks focus on small sets of very specific key management commands.

Survey of flaws III

- MAC command attacks:
  - Brincat-Mitchell (2001) – attacks using intermediate MAC output values;
  - Mitchell (2003) – attacks using variable truncation of MACs;

these attacks make use of specific MAC computation/verification commands.

Known problems

- Historically, it would appear that problems have arisen with flaws in small sets of closely-related commands.
- That is, problems involving interactions between unrelated commands appear to be rare.
- That is, most known problems can be fixed by designing individual commands more carefully, rather than worrying about unintended interactions.

What does this mean?

- Various possible interpretations of this observation.
  - Is there a big problem after all – perhaps we just need to design individual commands more carefully?
  - Pointers towards better design – get the individual commands right first!
  - It may be helpful to define the notion of separation of small command sets (modularity), i.e. when designing APIs to specify small sets of commands with no significant interactions between sets.

Agenda

1. Introduction
2. Security APIs – what is the problem?
3. New problems
4. Encryption and integrity – a case study
5. Rules for designing better APIs
6. Concluding remarks

Where should we look next?

- Past experience suggests we should look vey carefully at the design of individual API commands (and small sets of related commands).
- This also suggests that, when looking for problems in existing APIs, we should first check for poorly designed individual commands (and/or closely related command subsets).

Special purpose commands

- One area that has clearly caused problems in the past are commands designed for very specific functions (e.g. PIN verification).
- Specific applications (e.g. for banking) have very particular requirements.
- Meeting these requirements is an obvious source of vulnerabilities.
- Indeed, Mannan-van Oorschot (2008) suggest changing the way PINs are used to avoid the known problems with PIN cracking.

Secure cryptographic primitives

- More generally, most security APIs include a set of basic cryptographic functions.
- The idea that these might give rise to problems is a little more surprising, since surely these are things we understand well?

Possible crypto issues

- One source of possible problems is that the understanding of how to use simple cryptographic primitives has changed dramatically over the last 10-15 years.
- This is at last partly as a result of the development of a complexity-based theory for cryptography.
- This theory has resulted in the use of RSA in slightly more complex ways (for both encryption and signature) and has also affected use of symmetric cryptography.

Encryption without integrity

- One major change has been to view any kind of encryption without simultaneous integrity protection as highly dangerous.
- Combining encryption with integrity is not only required in order to achieve proofs of security, but over the last few years a wide range of attacks have been devised against schemes using just encryption (or poorly designed combinations of encryption and integrity).

Error oracle attacks

- One general class of attacks on encryption without integrity is made up of *error oracle* attacks (see *Proc. ISC 2005*).
- If an attacker can manipulate ciphertext, and repeatedly insert manipulated ciphertext into a channel, then the presence or absence of error messages (e.g. whether decryption is successful, why decryption failed, or whether the plaintext causes errors) can reveal information about the plaintext.
- In general, this problem cannot be removed, since errors may arise in higher level protocols, completely independent of the layer at which encryption is performed.

Encryption functions in APIs

- In many cases, symmetric encryption and MACing are implemented as separate functions in a security API.
- This may allow a variety of possible attacks, even if the application using the API never uses encryption without integrity protection.
- (We return to this issue a little later).

Order of cryptographic operations

- Another major change in view arising from developments in crypto theory has been regarding the order in which encryption and integrity protection are applied.
- Theory says that it is safer to encrypt first and then MAC protect (must verify MACs before attempting decryption, and **do not attempt to decrypt if MAC verification fails**).
- It is possible to achieve proofs of security for the 'MAC then encrypt' model, but such schemes are more complex and more at risk from side channel attacks (since decryption may fail prior to MAC verification, which could either trigger error messages or cause an early abort of processing).

How is this relevant here I?

- Error oracles (or other side channels) could arise in many ways from use of security APIs, e.g.:
  - Security APIs often include ‘translation’ commands, where data is input encrypted under one key and output encrypted under another – it may be possible to use such a command as an error oracle.
  - If encryption and MACing implemented separately, then, even if an attacker does not have direct access to the API, it may be possible to extract information from timing of events and/or error messages.
  - For key management commands involving managing encrypted keys, even if encrypted keys are also MACed, if the MACing is done on plaintext keys, then error messages may provide a side channel.

Relevance II

- Given the limited data storage of many security modules, it may be necessary to make multiple calls to the API if performing a crypto-operation on a large data set.
- Such multiple calls are generally implemented using special purpose ‘partial’ computation commands (or options to a single command).
- Two solutions:
  - temporary stored state inside the module (which can break design criteria);
  - Export from module of ‘partial’ results (which can then be input with next call to the API, along with further data for processing) – this type of solution is extremely hazardous as it breaks all assumptions about the cryptographic function.

Relevance III

- The issue of multiple calls also relates to a long-standing issue with some APIs supporting signatures.
- In some past designs (and perhaps even now) the hashing process necessary to compute a signature has been performed externally to the API, and the API takes a hash-code as input.
- This is incredibly hazardous, as it means that part of the signature function is implemented externally to the secure environment, which breaks any security proof for the signature scheme.
- This issue can be mitigated by ‘double-hashing’, i.e. hashing the data externally to the module and then performing the complete signature function (including the hashing) internally to the module (not clear how widely such an approach is adopted).

Relevance IV

- Other ‘obvious’ measures should be taken to properly limit scope of commends (although not always taken in past), e.g.:
  - MACs should be verified by recomputing the MAC and comparing internally to the module, and then outputting a ‘yes/no’ answer;
  - triple DES should be implemented internally, and not as three calls to a single DES function;

Agenda

1. Introduction
2. Security APIs – what is the problem?
3. New problems
4. Encryption and integrity – a case study
5. Rules for designing better APIs
6. Concluding remarks

IPsec in encryption-only mode

- Paterson and Degabriele have recently described and demonstrated attacks on IPsec when used in encryption only mode.
- These attacks apply to widely used IPsec implementations which conform to the relevant IETF RFCs (see *Proc. IEEE Security and Privacy 2007*).

Why look at this example?

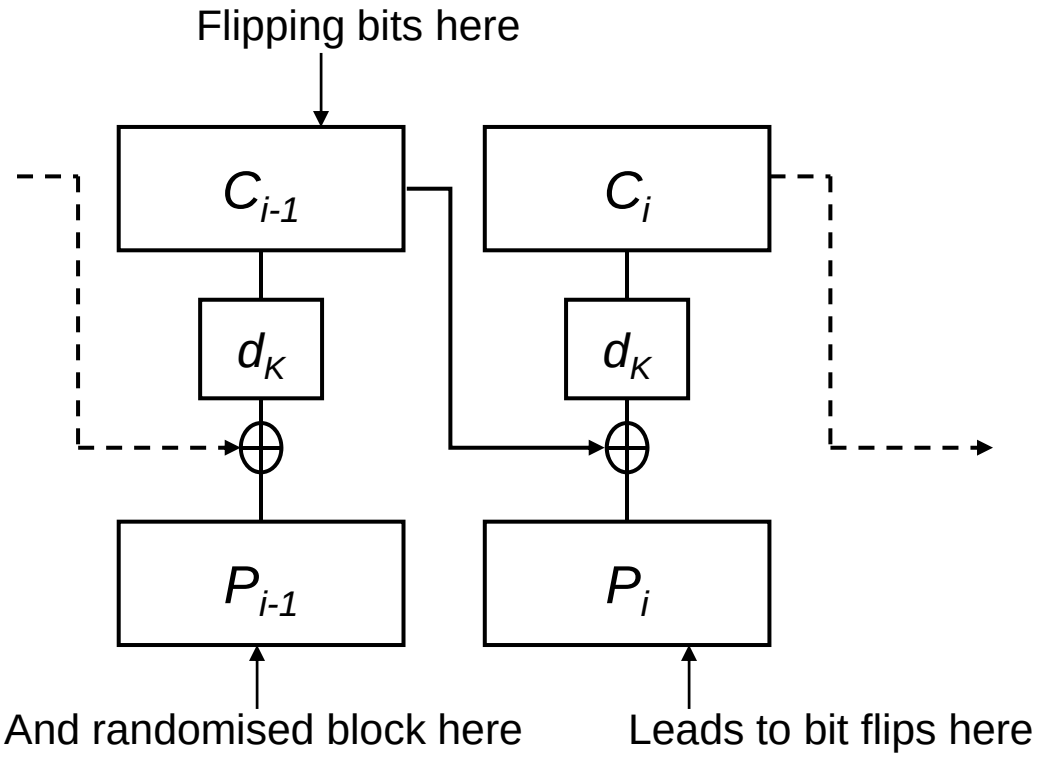
- This provides an example of the type of attack that is possible if an error oracle exists (and how error oracles can arise).
- It also illustrates the possible problems arising from the (complacent) belief that there is no need to follow crypto ‘best practice’.
- Such lessons almost certainly apply to many existing security APIs, which do not implement current best practice.

Main ideas

- Extension of Vaudenay's *padding oracle* attacks on CBC mode (*Eurocrypt 2002*) combined with Paterson-Yau techniques (*Eurocrypt 2006*).
- A padding oracle (a special type of error oracle):
  - attacker sends a ciphertext and learns only whether or not the underlying plaintext was correctly padded.
- Vaudenay showed that a padding oracle can be "leveraged" to build a decryption algorithm:
  - for CBC mode encryption;
  - for certain padding methods.

Bit flipping in CBC mode

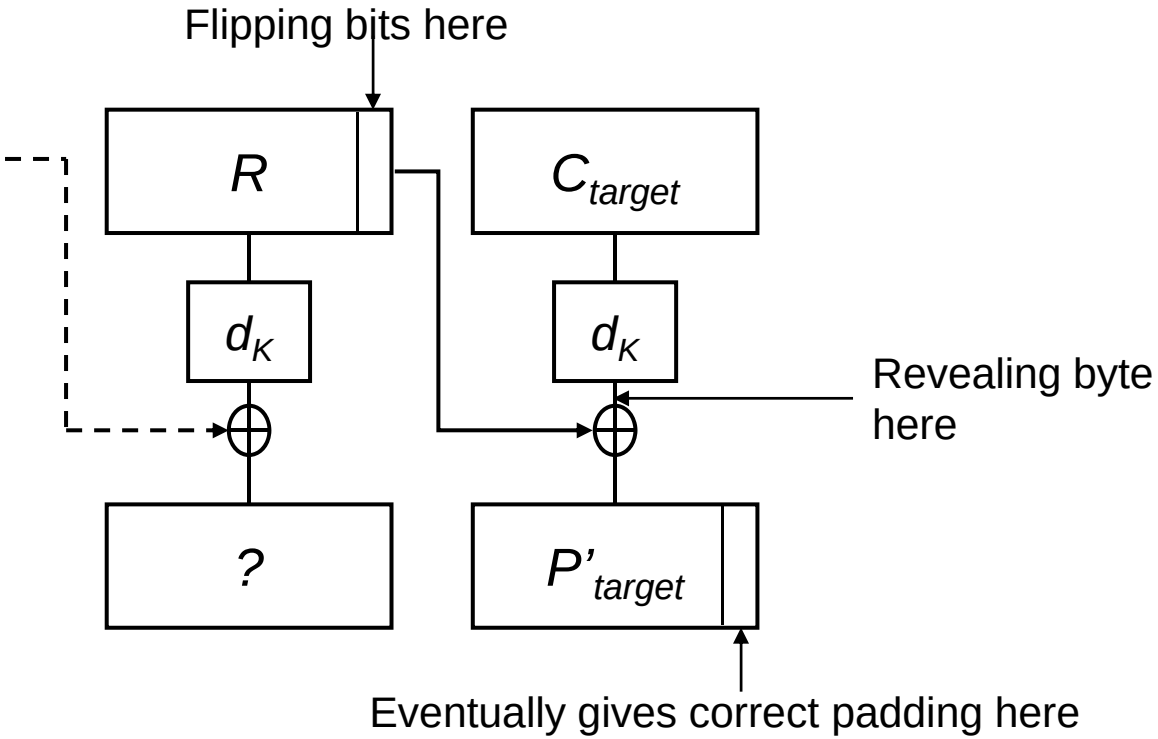
- Flipping bits in ciphertext block C_{i-1} leads to controlled changes in plaintext block P_i.
- But block P_{i-1} is randomised.



Padding oracles – a toy example

- Suppose only requirement for correct padding is last byte be “01” (hex).
- Repeatedly flip bits in last byte of *R* and submit to padding oracle.
- Padding oracle says “yes” iff:

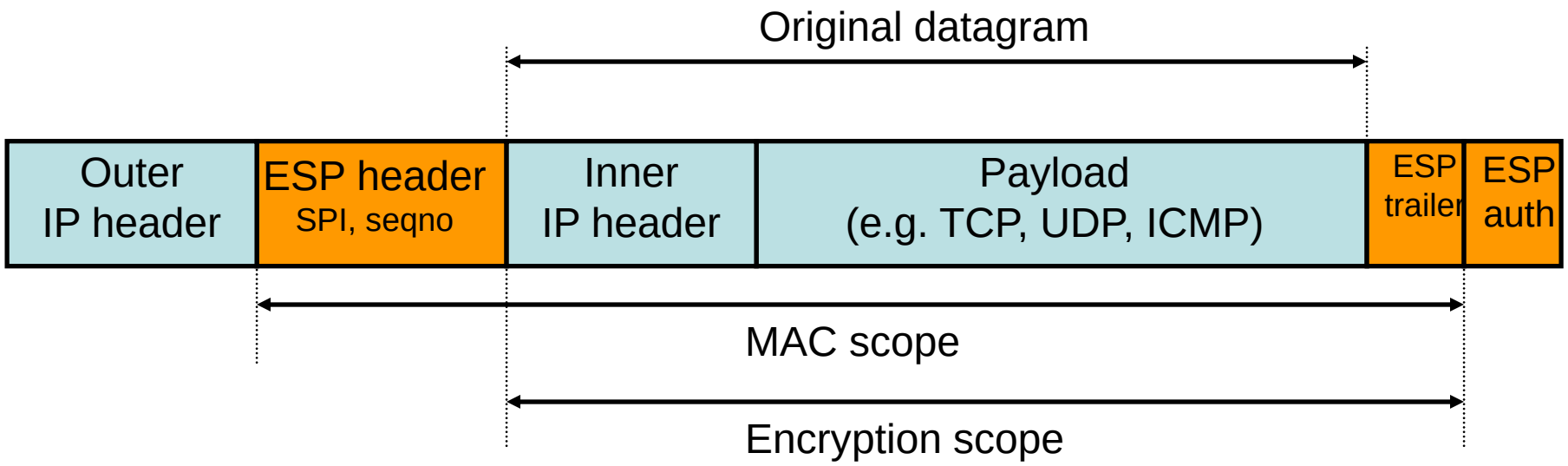
  Last byte of $R \oplus$ last byte of $d_K(C_{target}) = 01$
- This reveals last byte of $d_K(C_{target})$ and so last byte of the original P_{target}.



ESP

- ESP = Encapsulating Security Protocol.
  - v1, v2, v3 in IETF RFCs 1827, 2406, 4303.
  - IPsec's "encryption workhorse".
- ESP provides one or both of:
  - Confidentiality for packet/payload (v1, v2, v3).
  - Integrity protection for packet/payload (v2, v3).
- ESP uses symmetric encryption and MACs.
  - Usually CBC mode of block cipher for encryption.
  - HMAC-SHA1 and HMAC-MD5 for integrity protection.

ESP in Tunnel Mode



ESP header and trailer

- ESP specifies header and trailer fields to be added to IP datagrams.
- Fields in header include:
  - Security Parameters Index (SPI).
  - Sequence number.
- Fields in trailer include:
  - Any padding needed for encryption algorithm (may also be used to disguise payload length).
  - Padding length.
  - Next Header byte.
  - MAC value (if integrity protection used).

History of encryption in IPsec

- ESPv1 (1995) provided no integrity protection.
  - Reliant on separate AH protocol to provide this.
  - Bellovin (95 and 96) sketched a series of attacks on ESPv1 without AH.
- Bellovin-Wagner attack:
  - Limited recovery of plaintext from TCP segments:
  - Requires ciphertexts matching 2^{24} chosen plaintexts.
  - Requires receiver to ignore encryption padding format.
    - attack fails if padding check carried out upon decryption.
  - Recovers last byte of plaintext from TCP segments if byte length is equal to 1 modulo 8.
  - Requirements can be reduced to 2^8 *chosen* plaintexts if variable length padding acceptable.

Bellovin's Attacks (continued)

- Bellovin's paper presents a collection of attack sketches and ideas.
  - Theoretically interesting, but no attacks demonstrated to work in practice.
  - Drew attention to need for integrity protection along with encryption.
  - Sufficiently serious to influence development of RFCs.

Integrity protection and ESPv2

- IETF response to Bellovin's attacks:
  - ESPv2 (1998) recommends receiver should check format of encryption padding.
  - Also includes integrity protection as an option.
  - But implementations must still support "encryption-only" mode.
- ESPv2 represents a compromise between improving security and maintaining backwards-compatibility.

Integrity protection and ESPv3

- ESPv3 (2005):
  - still allows encryption-only ESP.
  - but no longer *requires* support for encryption-only.
  - gives strong warnings about Bellovin-Wagner attack and refers to theoretical cryptography literature to motivate need to use integrity protection.
  - *"ESP allows encryption-only … because this may offer considerably better performance and still provide adequate security, e.g., when higher layer authentication/integrity protection is offered independently."*

IPsec in **theory** and practice

- The theoretical cryptography community is well aware of the need to carefully combine integrity protection with encryption to prevent active attacks against encryption.
- Plenty of high-profile, real-world examples:
  - Kerberosv4, IEEE 802.11b, SSH, OpenSSL,…
- It is also well-known amongst IPsec experts that encryption-only configurations should be avoided - clear warnings against their use in the RFCs.
- So is there really any problem?

IPsec in theory and **practice**

- Developers are required by RFC 2406 to support encryption-only ESP.
- Developers rarely pass RFC warnings to end users.
- Developers don’t properly implement RFCs.
- End users don’t read RFCs or technical papers.
- End users might reasonably assume that encryption on its own gives confidentiality.
- Many on-line tutorials do not highlight the dangers of encryption-only IPsec.

IPsec in theory and practice

- From the IPsec Tunnel Implementation administrator's guide of a well-known vendor:
  - “If you require data confidentiality only in your IPSec tunnel implementation, you should use ESP without authentication. By leaving off the authentication service, you gain some performance speed but lose the authentication service.”

  http://www.cisco.com/en/US/docs/security/security_management/vms/router_mc/1.3.x/user/guide/U13_bldg.html#wp1068306 (last accessed 28/1/2008).

ESP trailer format

- Append a byte pattern of the form:

  01 02 … *y*
- Append the PL byte (*y* again).
- Append the NH byte (04 in tunnel mode).
- So valid ESP trailer formats are:

  00 04,

  01 01 04,

  01 02 02 04,

  01 02 03 03 04, …

ESP trailer oracles

- An *ESP trailer oracle* could be exploited to perform decryption.
  - This is an oracle telling the attacker if a trailer's format is valid or invalid.
  - Initial 2-byte pattern "00 04" implies 2^{16} calls to the oracle are needed to extract first 2 bytes.
  - 2^8 calls per byte for remaining bytes of each block.
- To make this work, we need to find a reliable ESP trailer oracle.

ESP Trailer Oracles

- Wrongly formatted ESP trailers should lead to packet drops.
  - Padding checks SHOULD be carried out:
    - but packet drop on failure is not mandated explicitly;
  - NH byte must equal 04 to pass policy checks
- So a packet drop would indicate an incorrectly formatted ESP trailer.
- But we also need an indication of when an ESP trailer is *correctly* formatted.

Building an ESP Trailer Oracle

- If ESP trailer is correctly formatted, then inner packet is eventually processed by IP.
- Paterson-Degabriele built a single tunnel mode packet that *always* results in an ICMP response when its inner packet is processed.
  - Capture a tunnel mode packet.
  - Modify TTL, Protocol or IP header length fields in inner packet by bit flipping.
  - Correct Header Checksum field by bit flipping.
  - One-time cost for construction.

ESP Trailer Oracle Attack

- For any target ciphertext block:
  - Splice random block and target ciphertext block onto end of ICMP-generating packet.
  - Inject this new packet into tunnel.
  - Target block interpreted as ESP trailer
    - ICMP message created if and only if ESP trailer correctly formatted.
  - ICMP message sent encrypted on reverse tunnel
    - Detectable by its length.
- This is an *ESP trailer oracle attack*.
  - Using behaviour of IPsec implementation at receiver coupled with presence/absence of ICMP messages as the oracle.

Implementing the RFC Attacks

- These RFC attacks work “in theory” against any IPsec implementation that strictly follows the RFCs.
- But many practical issues may interfere with the correct operation of the attacks.
- Are any implementations sufficiently strict?
- And what happens in reality?
- Look at open source implementations...

Implementing the RFC attacks

- **Linux**:
  - Comment in source code:
    ```
    /* ... check padding bits here. Silly. :-) */}
    ```
  - No padding check implemented.
  - So the RFC attacks don't apply because of incorrect implementation ... but then vulnerable to Bellovin-Wagner attack from 1995!
  - Also vulnerable to:
    - a variant of the RFC attack which can efficiently extract two bytes per block, implemented as a proof of concept;
    - another Paterson-Yau attack.

Implementing the RFC attacks

- **KAME, OpenBSD, FreeBSD, NetBSD, MacOS X**:
  - Crude padding check: check if pad length byte is 0 or if pad length byte = last byte of padding.
  - Not rigorous enough for the RFC attacks to work.
  - But a variant of the Paterson-Degabriele RFC attacks extracts three bytes per block for 2^{16} effort.

Implementing the RFC attacks

- **Openswan, strongSwan, FreeS/WAN**:
  - Don't allow selection of encryption-only configurations (despite mandated support in ESPv2).
  - All check padding carefully, but then don't drop packet if it's incorrect!
  - (RFCs don't explicitly mandate drop, but then what's the point of doing the check?)
  - So the RFC attacks won't work, but Bellovin's attacks will.

Implementing the RFC attacks

- **OpenSolaris**:
  - 3 different levels of padding check can be selected.
    - No check, KAME-style check, full padding check.
  - But the full check was incorrectly implemented!
  - Paterson and Degabriele reported the bug to Sun.
  - Sun fixed it in Release 55 of OpenSolaris.
  - After which, they successfully attacked the OpenSolaris implementation.
  - Attack complexity in line with theoretical results.
    - Dominated by 2^{16} trials to extract last 2 bytes of each block.

Summary of attacks

- There is a range of attacks against encryption-only ESP that work:
  - against any implementation strictly following the RFCs, e.g. OpenSolaris;
  - against many implementations not following the RFCs, e.g. Linux.
- The attacks that work in practice shouldn't work against the RFCs.
- The attacks that work against the RFCs often don't work in practice.

Discussion I

- Encryption-only ESP is dangerously weak in a very practical sense.
- No security is gained from provision of upper layer integrity protection, despite claims to contrary in ESPv3:

  ESP allows encryption-only … because this may offer considerably better performance and still provide adequate security, e.g., when higher layer authentication/integrity protection is offered independently."

Discussion II

- Attacks reveal poor lines of communication in the IPsec community in its widest sense.
  - Configurations known to be weak to IPsec insiders are still allowed in the standard.
  - These configurations get deployed by end-users.
  - 10 years on, many IPsec implementations don't follow advice given in standards anyway.
  - Why is that? What can we do to address it?

Discussion III

- Ultimately RFCs are standards for interoperability, but this causes problems for RFCs concerned with security.
  - Applying patches to security in each new revision is not the answer.
  - Such standards should **take a more conservative approach and adopt defensive designs**.
  - Despite the many real-world constraints imposed on standards development process.

Discussion IV

- Attacks reveal disconnect between theory and practice in cryptography.
  - Need for strong integrity protection well understood in theoretical cryptography, but not so well by practitioners and users.
  - Cryptographic implementation details are vital for security, but are not currently considered in theoretical security models.
  - Strong anecdotal evidence suggests these points apply equally to security API designs.
- Unfortunately, the gulfs between cryptographers, users of cryptography, and implementers appear to be growing.

Discussion V

- Implementers seem likely to ignore implementation requirements specified in standards if the reason for them is not blindingly obvious.
- One way of avoiding this problem is to design APIs to protect implementers against their own ignorance.
- That is, we **should design security APIs to only permit safe use of cryptography**.

Agenda

1. Introduction
2. Security APIs – what is the problem?
3. New problems
4. Encryption and integrity – a case study
5. Rules for designing better APIs
6. Concluding remarks

Research directions

- Current research includes work directed at the following problems:
  - Establish models within which properties of security APIs can be established.
  - Design compact APIs which can be reasoned about.
  - Find new ways in which APIs can go wrong (analyse existing APIs).

Is this sufficient?

- Work of all these types is clearly necessary.
- However, is it enough?
- One problem with designing ‘good’ general purpose APIs is that there is likely to be a need to add specific commands to existing APIs on an ad hoc basis.
- Customers for security modules often have very specific needs, which cannot be met using a ‘general purpose’ API.

Special purpose APIs

- Apart from the need to add specific commands, there is also a requirement for new APIs to meet very special requirements.
- One example is provided by trusted computing's TPM.
- That is, whilst clearly helpful, specially designed general purpose security APIs will never be sufficient for all needs.

Formal security proofs I

- Of course, in theory at least, we could just design our additions to existing APIs in such a way that security proofs can still be established.
- This may present formidable problems for suppliers of security modules, but of course we must try to develop means of analysing range of command types.

Security proofs II

- Interesting to observe that complexity-theory approach to cryptography fits rather nicely to the way APIs work in practice.
- That is, the theory assumes access by an adversary to a set of oracles, and proofs are formulated in terms of minimising the probabilistic advantage of the adversary.
- Access to an API is essentially access to a specific set of oracles.

Security proofs III

- Existing complexity theory approach should be capable of analysing the security of individual commands (and small sets of related commands, e.g. encrypt/decrypt).
- Analysis must take into account the set of error messages associated with commands (indeed, error messages could well be one of the trickiest areas to get correct).
- Would be interesting to also analyse multi-call commands to address large data sets.

Security proofs IV

- It would also be extremely helpful to find a way of modelling the notion of 'independence' of sets of commands.
- This would potentially enable subsets of an API's command set to be analysed separately.

Strong key separation I

- The need for **strong** key separation has been well-documented (notably by Clulow (2003) – see also Mitchell (2003)).
- Strong key separation means that the API should force keys to have just one use, with just one algorithm, and with **all** aspects of the algorithm fixed (e.g. MAC length).

Strong key separation II

- Strong key separation is implicit to mathematical models of cryptography.
- However, this fact does not appear to be universally recognised, either as:
  - an issue by the crypto community, or
  - a requirement by the crypto user community.

Other key management issues

- The risks of encrypting data that is not secret (e.g. and then using the same key for PIN protection).
- Issues arising from use of Live versus Test keys.
- Issues arising from flexible start and end points for encryption within a long message or PIN block.

Agenda

1. Introduction
2. Security APIs – what is the problem?
3. New problems
4. Encryption and integrity – a case study
5. Rules for designing better APIs
6. Concluding remarks

Where are the problems?

- Right now, most of the issues appear to arise from poorly designed single commands (or small sets of commands).
- Fixing these problems would appear relatively straightforward.
- (There are also certainly major opportunities for finding issues in existing security APIs).

New attack ideas

- One major possible direction for future attacks on APIs is to take advantage of 'old fashioned' view of crypto inherent in many of today's security APIs.
- For example, even the TPM v1.2 specifications from the TCG, which are quite recent, do not use cryptography in ways recommended by well-established theory.

Implementation issues

- A rather different issue is probably worth mentioning – namely that of the robustness of the internal code of a security module.
- The software development community is now familiar with buffer overflow problems arising from poor checking of input data – has this lesson been learnt by security module manufacturers?
- What about ‘old’ modules?
- Would it be worth fuzz-testing modules?

Thoughts for the future

- Bespoke additions to APIs will not go away, and so research must take account of this.
- Crypto commands should implement provably secure versions of crypto primitives, such as:
  - don't offer 'encryption only' commands;
  - use provably secure variants of RSA, including padding and redundancy.
- More generally, don't introduce unnecessary flexibility, and avoid 'optimisations' at expense of implementing well-defined primitives.

The final slide!

- Thanks to:
  - organisers for inviting me to come and speak;
  - Kenny Paterson for allowing me to borrow some of his slides on the IPsec attacks;
  - Mike Ward for a number of valuable comments and corrections;
  - you for listening.
- Questions?

  (by all means contact me offline – e.g. at c.mitchell@rhul.ac.uk – for detailed references to the various work mentioned).