How secure is your VPN software really?

Virtually Secure VPNs

The security of Virtual Private Networks (VPN) is often more virtual than real. Beyond the big three (SSL, SSH and IPsec) we find lots of applications with serious flaws. This article explains the details and recommends some solutions. PETER GUTMANN

There are quite a number of open-source (as well as proprietary) VPN applications around, but how secure are they really? In practice there's a lot more to creating a secure VPN than simply bolting some Blowfish encryption and RSA key exchange onto a network socket. This article looks at common pitfalls in creating VPN implementations, and contains guidelines and recommendations for people wanting to create their own VPN software or add VPN functionality to an existing application.

Leaving aside the more obscure requirements, even the basic three mentioned in Box "VPN security requirements" are quite tricky to get right. Figure 1 shows a generic template for a secure VPN implementation, consisting of a black-box handshake to the left and the actual data transfer portion to the right. The handshake can be anything appropriate (TLS, SSH [11], even IPsec's IKE [12]), with the keying material obtained from the initial handshake phase being passed on to the VPN data transfer phase.

Each packet is being given confidentiality protection (inner wrapper, using triple DES), message integrity protection (middle wrapper, using HMAC-SHA1) and traffic flow integrity protection (outer wrapper, using IPsec's sliding-window algorithm).

THE AUTHOR

Peter Gutmann is a researcher in the Department of Computer Science at the University of Auckland working on the design and analysis of cryptographic security architectures. He helped write the popular PGP encryption package, has authored a number of papers and RFCs on security and encryption including the X.509 Style Guide for certificates, and is the author of the open source cryptlib security toolkit and the book "Cryptographic Security Architecture Design and Verification" (Springer-Verlag, 2003). In his spare time he pokes holes in whatever security systems and mechanisms catch his attention and grumbles about PKIs.

Confidentiality

Let's look at some common pitfalls that people run into when creating VPN implementations without following the template. Some VPN protocols use RC4 as their encryption algorithm. Starting with Windows 3.1 and going on through to the late 1990s, Microsoft really liked using RC4 everywhere, and managed to get it wrong almost every time they used it, which is why they eventually stopped doing so. However, non-Microsoft apps like ECLiPt *http://freshmeat.net/projects/ecliptsecuretunnel/* and mirrordir *http://mirrordir.sourceforge.net/* still use it.

RC4 is a simple stream cipher that generates a pseudorandom output (the key stream) which is XOR'ed with the plaintext data to encrypt it. At the other side, you generate the same key stream and XOR it with the encrypted ciphertext data to recover the plaintext. It's simple to use, which is what made it so popular.

The problem with this is that XOR is commutative. Using the key stream and ciphertext, we can recover the plaintext. However, using the plaintext and ciphertext (for example some email we sent to the VPN user, or an e-commerce transaction that we sent to their web server and observed the VPN traffic that it corresponds to) we can recover the key stream. We can then XOR the recovered key stream with other VPN packets where we don't know the plaintext to recover their contents.

Even if we can't inject plaintext, by XOR'ing two encrypted packets together we can recover the XOR of the plaintexts because the key stream cancels out. At one point Microsoft attempted a sleight-of-hand "fix" for one of their RC4 appli-

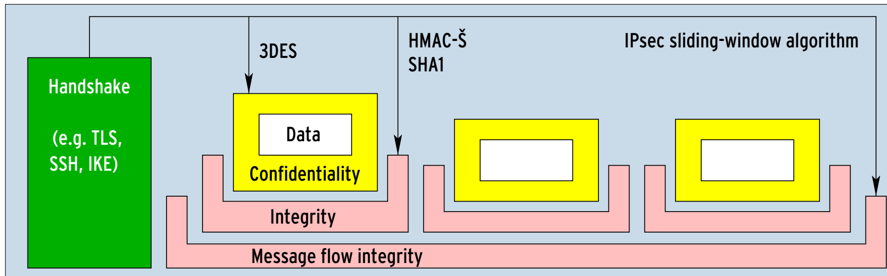

Figure 1: A secure VPN implementation consists of a handshake (e.g. TLS, SSH and IKE) which obtains the keying material for encryption (3DES) and integrity protection (HMAC-SHA1) of the actual data transfer

cations by switching from a 32-bit to a 128-bit RC4 key, which ignored the fact that the key length was irrelevant because all you needed to do was XOR two packets together to recover their contents.

Trivial attacks on RC4

One particularly trivial attack is possible on a challange/response protocol using RC4: By XORing the challenge and encrypted (or decrypted) response, we can recover the key stream (802.11 vendors quietly abandoned this part of WEP when someone pointed out what it was they were doing). RC4 also has some minor cryptographic weaknesses (some of which have been known for many years) that are a bit too complex to describe here, but that make its use in new designs questionable unless special precautions are taken. Finally, RC4 lets an attacker modify messages in arbitrary ways, as described in the next section.

The alternative to a stream cipher is a block cipher, of which the best-known is DES and its stronger variant triple DES (3DES). Other block ciphers are the more recent AES, IDEA (from PGP 2), Blowfish (from Bruce Schneier's book "Applied Cryptography"), and CAST (from PGP 5). There aren't any major external differences between them, triple DES is the conservative choice (it's the most heavily-analysed, has been around forever, and is supported by virtually everything), while AES is its successor and is much faster but also much newer, so it hasn't stood the same test of time that 3DES has. Although newer algorithms like AES were designed with speed of operation as a major design goal, it's all relative: Even an old 1GHz CPU can run 3DES at around 50Mb/s, and AES at around 100Mb/s. The Free S/WAN folks have further performance figures [1]

A block cipher, as the name implies, encrypts an entire block of data, usually 64 but more recently 128 bits at a time. This makes it rather harder to use than RC4, which does a byte at a time, since you need to work around the fact that most messages aren't an exact multiple of 64 or 128 bits long. When DES was standardized, a variety of modes of oper-

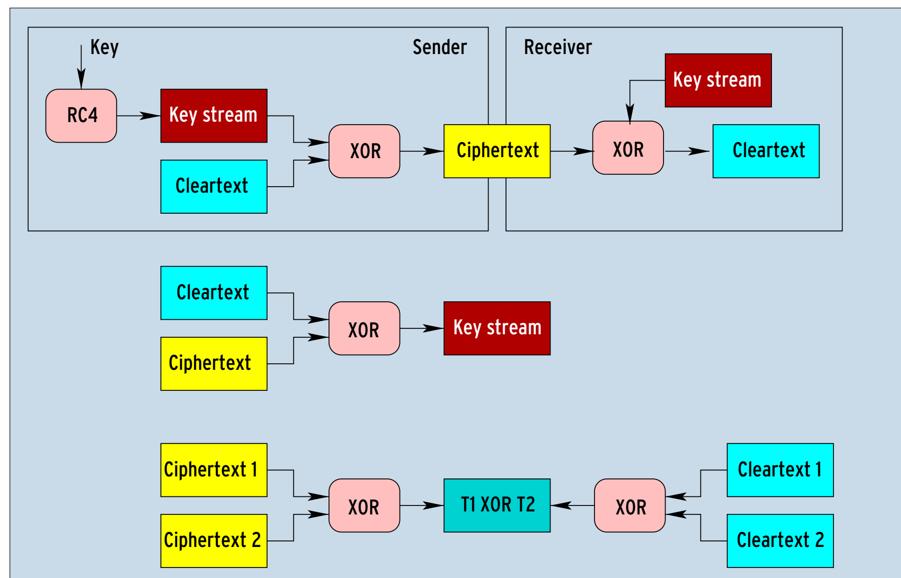

Figure 2: RC4 can be dangerous if used incorrectly. Here, the sender generates a key stream and XORs it with the plaintext (top). Injecting plaintext and reading the ciphertext is enough to recover the key stream (middle). By XOR'ing two encrypted packets together we can recover the XOR of the plaintexts

VPN security requirements

A VPN needs to meet a number of requirements that go above and beyond those of the core Internet protocol suite. These security requirements include:

Confidentiality: An attacker shouldn't be able to determine message contents. Providing this goes a long way beyond simply encrypting the data - an attacker may be able to cause harm simply from knowing the size of the encrypted data, or observing bit patterns in encrypted data, or being able to recover a single bit of data (for example a boolean flag).

Authenticity and access control: An attacker shouldn't be able to feed you data that appears to be from a legitimate user. Again, this can get tricky – the attacker could replay an authentic message from a genuine user, which you need to be able to detect.

Integrity protection: An attacker shouldn't be able to modify message contents. An extended form of this is traffic flow integrity protection, in which an attacker shouldn't be able to insert/replay ("Pay $10,000 to my account"), delete ("Look out, someone's modifying our messages!"), or reorder ("rm backup", "mv valuable_data backup") messages. Note that an attacker doesn't have to defeat any of the message confidentiality, authenticity, or per-message integrity measures to perform many types of traffic flow modification attacks.

Availability: A variety of other, lesser requirements such as DoS-protection (Denial of Service) also exist. These are frequently outside the control of the security layer (e.g. SYN-flooding is handled at a very low level in the IP stack). Providing DoS protection in the security code is also a good idea because many security protocols perform quite heavyweight crypto operations for which it's possible to overwhelm a server by firing a large number of client connection messages at it. It doesn't cost the attacker anything to generate these (they just contain random garbage), but it requires an expensive public-key crypto operation (e.g. RSA or DH) at the server to find out that they're just garbage.

ation were defined for it, with various properties such as hiding data patterns and allowing byte-at-a-time operation. Of these, the least secure is electronic codebook (ECB) mode, which encrypts independent 64-bit blocks and is warned against in every book on encryption. ECB mode is used by vtun, *http://vtun.sourceforge.net/*.

Modes of operation

Since each block is independently encrypted, a given piece of plaintext always encrypts to the same ciphertext, so that if you send in a data block ABCD and observe that it encrypts to WXYZ once it's on the VPN, you know that in the future any encrypted block WXYZ that you see would decrypt to ABCD. This allows you, by simple trial and error, to determine portions of someone else's message contents without knowing the key (it's like encrypting Unix passwords without using a salt). Even worse, since the blocks are independent, you can perform neat cut-and-paste attacks as described in the next section. For a cryptographer, seeing encryption software that uses ECB mode is a danger sign somewhat akin to seeing Visual Basic and HTML listed under "programming skills" on someone's resume.

There are other modes that don't have these problems, of which the most widely-used is cipher block chaining mode (CBC). CBC (and other modes like CFB, ciphertext feedback) make block *n* depend on block *n-1*, and use a random -*1'th* block (the initialization vector or IV) to ensure that the first block is randomized as well. These modes are illustrated in Figure 3. This means that even repeatedly encrypting the same data with the same key (but different IV) produces a different output each time, making the trivial attacks on ECB mode described above rather more difficult, although not totally impossible – the birthday paradox says that with a 64-bit block cipher after encrypting 2^32 blocks you're likely to see a repeated block, which is why AES went to a 128-bit block size.

Certificational Weaknesses

This type of weakness is known to cryptographers as a certificational weakness. It's not terribly likely to be exploited in practice, but if you have a choice of two algorithms or modes and one exhibits the problem and the other doesn't, it's better to choose the one that's immune. At some point in the future someone

Birthday paradox

How many people do you need to have in a room to ensure that the odds of two of them sharing the same birthday are better than 50%? The chances of someone having the same birthday as you are a rather unlikely 1/365 or 0.3%. However, the chances of two arbitrary people sharing the same birthday increases much more quickly than 0.3% per person, because each person present removes their birth date from the pool of available dates.

This means that the range of possible choices drops from 1/365 to 1/364, then to 1/363, and so on. Once you get more than 23 people together, the chances of clashing birthdays is over 50%. This paradox has implications in the safe choice of block sizes for block ciphers and MAC algorithms.

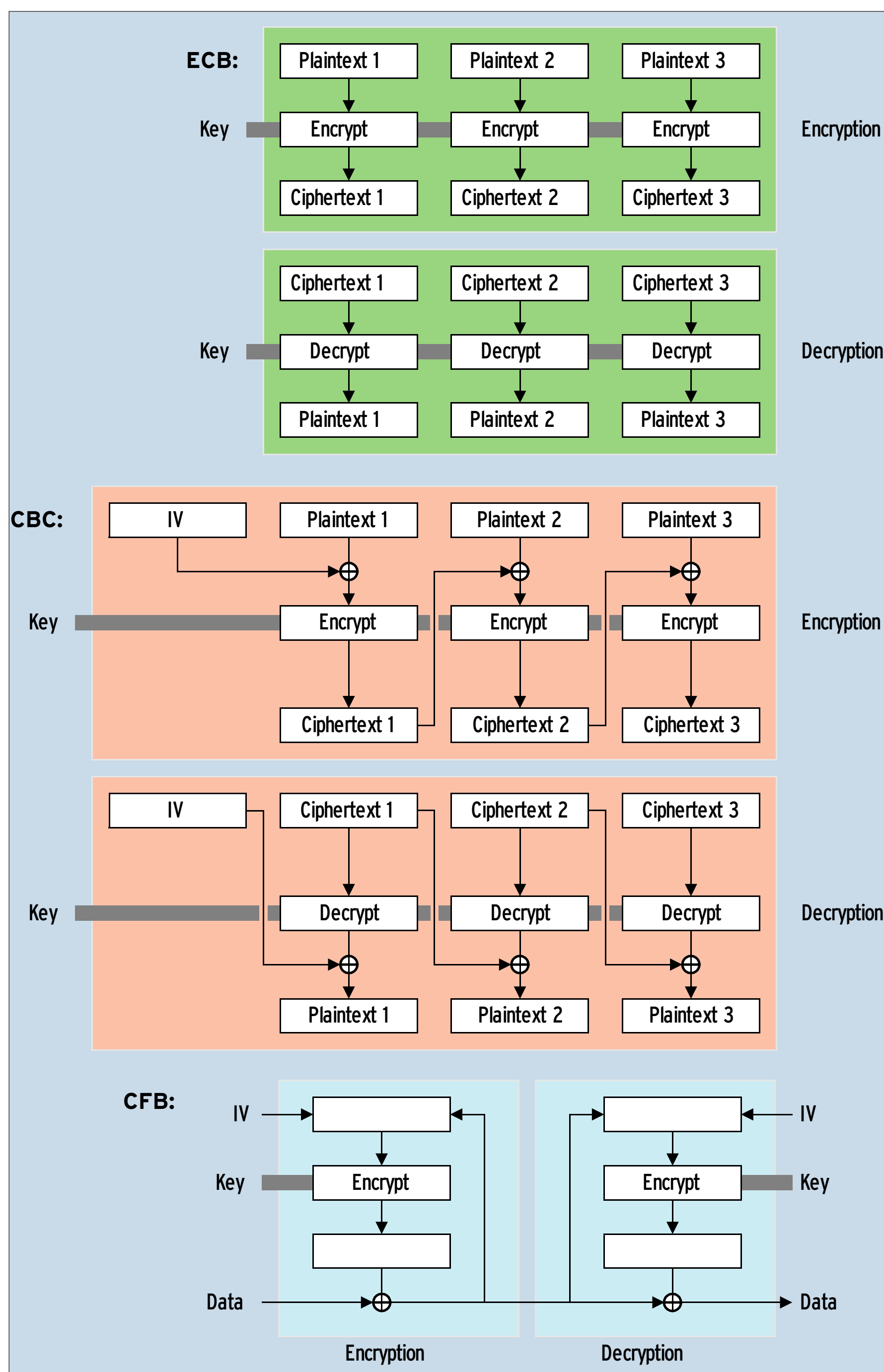

Figure 3: Block cipher encryption modes. Both CBC (middle) und CFB (bottom) make each block depend in its predecessor. In contrast, ECB (top) encrypts every block independently

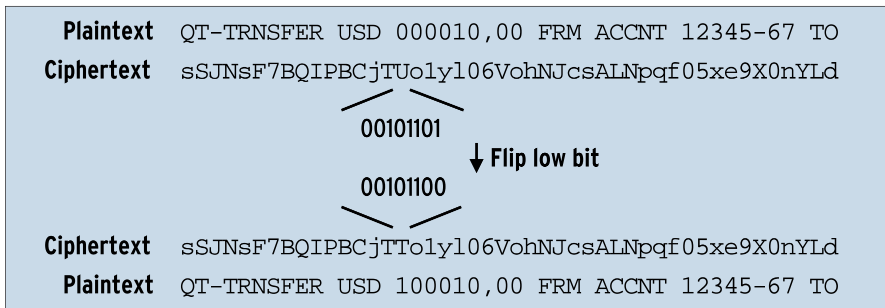

Figure 4: Flipping a single bit of RC4-encrypted data can have a major impact: Instead of $10, the modified EDI transaction now transfers $100,010

may figure out how to exploit it. Quite a number of real-world attacks on security protocols from the last few years were regarded as certificational weaknesses up until about the time that the attack details were published. This is why protocol designers prefer to over-engineer security protocols to be resistant to attacks that are no more than a glint in some cryptographer's eye: They don't want to have to re-deploy their entire installed base a year or two later when someone figures out how to make the attack practical.

There are a few other potential problems with IVs as well. For one thing, you have to actually use them for them to be effective (TunnelVision *http://open.nit.ca/wiki/?page=TunnelVision* and Zebedee *http://www.winton.org.uk/zebedee/* don't). In addition the IV has to be random, and unpredictable to an attacker [2] [3]. SSL and SSH, used the last block of packet *n-1* as the IV for packet *n*, which means that an attacker who observes packet *n-1* can tell in advance what the IV for packet *n* will be. The next version of SSL, TLS 1.1, will fix this (the IPsec folks knew about this one years ago, so IPsec doesn't have the same problem [4]). This is another certificational weakness – so far no-one's found a way of doing any really serious damage with it, but that doesn't necessarily mean that it's safe to ignore.

Message integrity

Some programmers assume that encrypting data provides integrity protection. Nothing could be further from the truth. It's often quite simple to arbitrarily modify encrypted data without knowing the encryption key. Consider for example RC4 as described in the previous section and as used in ECLiPt, mirrordir and any number of Microsoft applications. In the example in Figure 4, flipping a single bit completely changes the meaning of the message, so that a standard EDI funds transfer that in its original form would have moved $10 into my account is now moving rather more across.

Bit flipping with other stream ciphers like CFB is similarly simple (several VPNs use RC4 or CFB mode without integrity protection).

Cutting and pasting

A previous section mentioned problems with ECB mode as used in vtun. Figure 5 shows how to break the security of a VPN using (say) triple DES without needing to know the decryption key.

Recall that each block is encrypted independently. To start, I go to a banking server and submit a transaction. Observing the VPN traffic, I see what this encrypts to. Later in the day, I see another message, so I cut and paste the blocks containing my account number into the message and now the money goes into my account, without me having to know the encryption key.

I'm not exactly sure at this point what the size of my windfall is going to be because the information is encrypted, so I'll have to wait until the payment appears in my account to see how much I've won. If it's not enough, I can repeat the process as often as required.

What about using CBC mode, in which each block depends on the preceding one? Some VPNs use CBC or other modes like CFB in a attempt to avoid the weaknesses in EBC. These modes are better than ECB, but only a little. CBC and CFB have a very useful (at least in some circumstances) resynchronization property in which messing with a block will garble the following one, but after that the cipher will recover. This means that even if you use these modes, you can still perform the cut-and-paste attack described above. In addition, CFB lets you undetectably modify the last block just as with an RC4-style stream cipher.

Providing proper integrity protection

The only way to provide any real integrity protection is to do it explicitly. With the exception of a few quite recent, still somewhat untried, and unfortunately mostly patent-encumbered encryption modes, none of the standard encryption modes can provide message integrity protection.

A first attempt at providing integrity protection would be to add a simple checksum like a CRC to the message (some VPNs don't even do that, using no protection at all). Of the ones that do, CIPE, *http://sites.inka.de/sites/bigred/devel/cipe.html* uses a CRC, as did SSHv1 before it. Unfortunately, this isn't secure. CRCs are reasonably good at detecting inadvertent modifications, but completely useless for detecting deliberate ones – you can create data that has any CRC you want it to have, including the one that the original message had if you want to create a forged message. Further-

Deposit	$10,000	in acct.	number	12-3456-	789012-3
H2nx/GHE	KgvldSbq	GQHbrUt5	tYf6K7ug	S4CrMTvH	7eMPZcE2
H2nx/GHE	5guZEHVr	GQHbrUt5	tYf6K7ug	Pts21LGb	a8oaNWpj
H2nx/GHE	5guZEHVr	GQHbrUt5	tYf6K7ug	S4CrMTvH	7eMPZcE2

Figure 5: Block ciphers in ECB mode are vulnerable to simple cut-and-paste attacks. Copying the bank account number from a regular transaction (top) into someone else's transaction (bottom) is all that's needed to redirect the funds

more, even encrypting the CRC won't help you. The details are a bit too complex to explain here, but it's covered in a 1998 Core SDI paper [5] (the use of CRC-32 was actually a certificational weakness in Kerberos for over a decade before it was actively exploited in SSHv1). This weakness lead to the SSHv1 insertion attacks and the more or less complete abandonment of SSHv1.

Integrity protection using MACs

In order to provide proper message integrity, a special-purpose cryptographic checksum called a message authentication code (MAC) is required. IPsec cargo-cult protocol design practice seems to favour a MAC size of 96 bits. IPsec truncated the MAC to 96 bits because that makes the IPsec AH header a nice size, with a payload length of exactly 128 bits (4 32-bit words); everyone else assumed that the number 96 had some magic significance and copied it into their own designs.

SSL/TLS and SSHv2 use a full-size 160-bit MAC. vpnd *http://sunsite.dk/vpnd/* uses a MAC, but truncates it to a mere 16 bits. On a T1 link, the birthday paradox says that you can expect to see your first MAC collision after just over half a second. As an optional alternative to the MAC, vpnd invents its own 16-bit checksum, a peculiar homebrew affair:

```
while(length--)
  sum=((sum<<1)|((sum>>15)&1))
      ^*data++;
```

with very poor properties (the checksum cycles so that after 16 bytes of repeated data the low 16 bits contain all ones; after 16 more bytes the input cycles back to all zeroes, and then repeats).

Traffic flow integrity

This is where things start getting a bit more tricky. Even if you've managed to get the encryption and per-message integrity protection right, an attacker may still be able to cause havoc by inserting/replaying, deleting, or reordering your otherwise-protected data packets. Almost none of the VPN apps provide any protection against this. The same mistake was made in very early versions of IPsec, which didn't have much traffic flow integrity protection because what was being tunnelled was IP traffic, and IP expects unreliable links and deals with them itself.

However, like the fact that a CRC is good for detecting inadvertent changes but hopeless for detecting deliberate ones, there are all sorts of nasty things you can do with deliberately-chosen "failures" rather than random, network-induced ones, because the protocols in question were never designed to handle deliberate, maliciously-applied "failures". Steve Bellovin (one of the IPsec designers) wrote a paper on this that contains a whole catalogue of problems [6]. This was subsequently fixed in IPsec, but not in the other VPN apps mentioned above.

False Assumptions

Related to this is a user perception problem: Users may expect that the use of a VPN with all manner of security features gives them stronger guarantees about the quality of service that they're getting than the use of an unprotected channel. This is not an unreasonable assumption to make, the whole point of a VPN is to provide additional guarantees that go beyond the basic IP ones. Alternatively, the user may be tunnelling something other than IP data (vpe *http://savannah.nongnu.org/projects/vpe*, for example, tunnels Ethernet frames) which doesn't even try to provide any traffic flow protection.

In this case the VPN author is expecting the user to provide the service, and the user is expecting the VPN author to provide the service. The solution to this problem is fairly simple: Include sequence numbers inside the secured envelope (Figure 6) and have the recipient arrange received packets by sequence number, re-ordering, discarding duplicates, and complaining about missing packets as required. This is simple enough for a TCP link (the packets have to arrive in order, so any duplicate, missing, or reordered packets are an indication of an attack), but requires quite a bit of extra work with the inherently unreliable UDP.

IPsec solves the problem with a sliding-window algorithm [7] that provides localized reliability at the IPsec level. An alternative would be to provide a reliability layer on top of UDP and then treat it as if it were a TCP link, with standard sequence-number-based protection as described above. Both of these options require a bit of thought (and careful coding) to get right.

Uncontrolled control channel

To compound the problem, several of the non-IPsec VPNs send session control data over the not-quite-secure tunnel that they've established, made worse by the fact that the control channel doesn't contain tunnelled IP data so even the minimal protection provided by the IP-level checking isn't present.

VPN control data is a lot more sensitive than payload data, since an attack on a single control-channel message can compromie every message sent over the data channel. This is why SSL, SSH, and IPsec all use a full protocol re-handshake

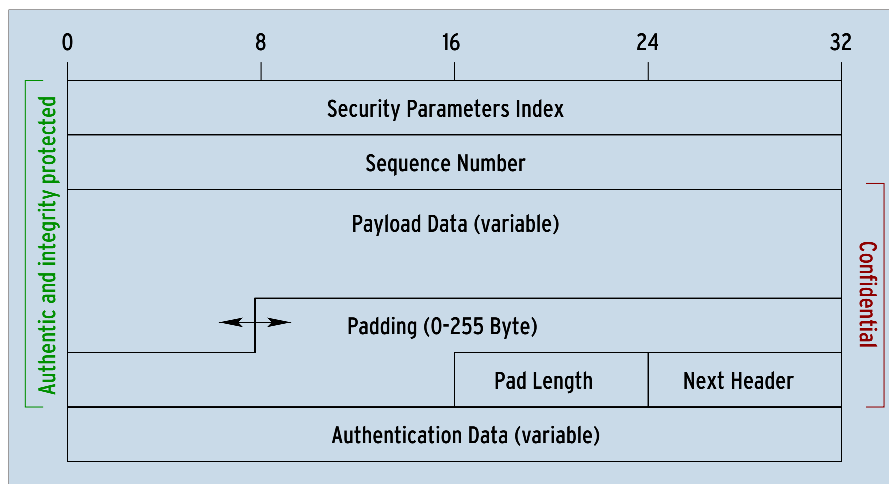

Figure 6: The ESP header (an IPsec protocol) illustrates some important security features. The sequence number is used for replay protection and the padding obscures the real payload length

Some pontificating

Designing a secure VPN protocol is hard. Really, really hard. I wouldn't feel comfortable doing it, and I do this sort of thing for a living. It took some of the world's best cryptographers and security engineers years to design IPsec because (all joking about IKE's design-by-committee process aside) every time they thought they had the design sorted out, someone would come along with a new attack on some part of the protocol. In some cases entirely new security criteria and analysis techniques had to be created in order to evaluate the security of various aspects of IPsec.

Many previously undreamed-of problems were discovered in the process, and were fixed in the final design. As a consequence of work on protocols like IPsec, SSL, and SSH, the standards for key exchange/agreement and authentication protocols have evolved considerably in the last few years. In particular, authenticated key agreement turns out to be surprisingly hard to do right (it's easy enough to *do*, it's just really hard to do *right*). Even the basic security models used to evaluate the protocols are still the subject of active research.

The reason why protocols like IPsec (without the IKE portion), SSL, and SSH are the way they are is because that's about the most minimal protocol you can create that's still secure.

to update cryptographic parameters rather than just pushing a new key or security parameter down the tunnel. Since the response to messages with bad checksums is frequently to drop the packet, it's possible to no-op out key-change messages (forcing continued use of a compromized key) by flipping a bit or two. Without replay protection an attacker can even replay messages containing instructions to switch to old keys, forcing the re-use of a compromized key.

Most of the VPN apps don't obscure the payload length with random padding, allowing an attacker to quickly identify key management packets and apply the above attacks. Obscuring the payload length is also useful to prevent an attacker from trivially identifying other fixed-length messages like TCP ACKs, ARP requests, and DHCP discover messages, or identifying encrypted ICMP messages/responses, decoding their contents, and injecting their own ICMP messages [8]. In order to obscure information about sensitive packets containing data like encrypted passwords, implementations of protocols like SSH often pad the packets out to a fixed length to make determining the payload length impossible, or send a flood of cover traffic to mask the presence of packets containing valuable information.

Disabling encryption

Another control-channel attack (which is really a message-integrity attack rather than a traffic-integrity attack) works against VPN apps that transport encryption keys in improperly-protected packets. The most obvious attack is to flip the bits so that you get an all-zero (or otherwise known/predictable) key. However, against some ciphers it's possible to perform an even simpler attack that merely requires the ability to flip a single bit. It doesn't matter what the value is, as long as it's different from what was originally sent. For this to work, the victim needs to be using DES or 3DES. This cipher has the property that the key bits are parity-checked, which was a consideration in the early 1970s when DES was being designed. According to the specification, you're not supposed to use a key if the parity bits are wrong, indicating that the key was corrupted in transit.

The attack works because many VPN apps don't bother checking return values for crypto functions, assuming that they always succeed. Examples of applications that do this are TunnelVision, vpe and Zebedee. What you do here is flip a bit in the encrypted key packet, and when the VPN software tries to load the key, the key load fails because the parity is wrong. Since the software doesn't check that the load succeeded, it ploughs ahead with an all-zero key. This isn't just a simple case of poor programming practice: Any properly-designed security application should never even get to the stage where it loads a tampered key, because the tampering should be detected well before that point (there are a pile of side-channel and related-key attacks that are made possible with this, the DES-key attack just happens to be the simplest one to implement).

Other problems

So far I've only talked about the basic VPN data and control channel processing, and not covered the initial handshake and authentication process. The reason for this is that it's an extremely complex subject (see "Further reading") that usually consumes multiple chapters of a security book and would consume a similar amount of space here, first to describe all of the potential avenues of attack and then to apply them to the various VPN implementations. The task is made more difficult by the fact that the majority of them use homebrew handshake mechanisms whose design has to be reverse-engineered from the source code.

In addition, a number of the implementations exhibit not just protocol design flaws but various implementation errors such as generating encryption keys from the process ID and time or via rand(), or performing little or no checking on input data for out-of-bounds values and range errors (the same problem was found in Microsoft's PPTP implementation, various attempts to exploit protocol flaws instead crashed

Pre-built VPNs

If you need secure off-the-shelf VPN software, the most obvious solution to go with is an IPsec implementation, typically Free S/WAN [12] *http://www.freeswan.org/*. Unfortunately, feedback from users indicates that this is quite difficult to use, even more so than IPsec applications in general. In the future the moving of Kame-derived IPsec functionality into newer Linux kernels should help this situation a bit.

Of all the non-IPsecVPN applications that I could find, only OpenVPN *http://openvpn.sf.net/* and Yavipin *http://yavipin.sf.net/* didn't have immediately obvious security problems (I'm not sure what that failure rate would imply for closed-source VPN applications). OpenVPN is built on SSL for the control channel (taking the place of IPsec's IKE), and uses a design closely modelled on IPsec's ESP for the data channel. Yavipin is an original design, but comes from someone who knows what they're doing and appears to be well-written. Both are pure user-space applications and have little of IPsec's complexity.

Rolling your own

If you really need to roll your own VPN software or want to add VPN- style functionality to an existing application, create the control channel using the expertise of the SSL and SSH developers, and the data channel using the expertise of the IPsec developers (this is exactly what one VPN application, OpenVPN, does). For the SSL/SSH portion there's OpenSSL *http://www.openssl.org/* and OpenSSH [11] *http://www.openssh.org/* or alternatives like cryptlib *http://www.cs.auckland.ac.nz/~pgut001/cryptlib/* (my own modest contribution) providing SSL, SSH, and a pile of other stuff.

For the IPsec portion, there's the Kame framework (in newer kernels) or something like ipsec_tunnel *http://ringstrom.mine.nu/ipsec_tunnel/* to retrofit to older ones. ipsec_tunnel is an extremely lightweight (38K of source code) IPsec ESP implementation, although it appears to be missing some functionality such as the sliding-window replay protection that would have to be added to provide proper security.

There is an effort under way to define standards for non-IPsec VPNs, the discussion is available online via Gmane *http://news.gmane.org/gmane.network.vpn.theory*.

the server because the code didn't do much checking of input data [9]). It would take a fairly lengthy and detailed code audit to catalog all of these problems, and when a particular flaw is found it's not certain whether it's a problem with the (undocumented) protocol design, or not actually part of the design but simply an implementation error.

Speed vs. security

Some VPNs offer multiple modes of operation, usually with one optimized for security and one optimized for throughput at the expense of security (this is entirely unnecessary, the small extra overhead for proper message and traffic flow integrity protection is mostly lost in the general VPN overhead). I'll refer to the throughput-optimized mode here as "Attack me first!" mode, because no attacker worth their salt will even look at the secure mode when "Attack me first!" mode is also available.

Here's how they do this: When the victim sets up a VPN in secure mode, the attacker bogs down the link using any one of 1,001 standard denial-of-service (DoS) techniques. Once the victim switches to "Attack me first!" mode, they remove the DoS. Obviously the problem was the use of secure mode, because as soon as the victim switched to "Attack me first!" mode, throughput improved. The victim is happy, and so is the attacker. This type of downgrade attack is a standard mechanism used against various types of Windows authentication, where you don't attack any of the newer mechanisms if you can help it because it's much easier to just convince the victim to switch to one of the totally insecure Windows 3.1-era modes (protocols like SSL include built-in protection against this type of problem).

An attack of this kind was reported on the SWIFT network (Society for Worldwide Interbank Funds Transfer, moving about $US6 trillion a day) in the mid-1980s, where link encryptors worked perfectly encrypting data sent over test links but any attempt to send live link data resulted in data corruption and lost packets. After weeks of trying, the bank gave up and sent its financial transactions in the clear, whereupon no more data corruption occurred.

A question of timing

Sometimes the problems that have to be countered are extremely subtle. For example, since security isn't closed under composition, two individually secure subsystems can combine to produce an insecure overall system (this one is a standard nightmare for protocol designers, and an area where a lot of recent research effort has gone). Consider a VPN that uses well-designed and correctly-implemented confidentiality protection and appropriate integrity protection. By themselves, they're secure. However, when combined, they can cause problems.

A standard approach to per-packet protection is to take your message, MAC it to provide integrity protection, and then encrypt the whole lot to provide confidentiality protection. To perform the attack, an attacker modifies the message in transit and observes what happens when it gets to its destination. If the message is rejected fairly quickly, the problem was detected during decryption. If the rejection takes a bit longer, the problem was detected after decryption by the MAC'ing. This allows the attacker to differentiate between attacks on the encryption and attacks on the MAC.

At the moment this is (mostly) just another certificational weakness, but that doesn't mean that someone won't discover a means of turning it into a real weakness ten minutes after this article appears (if they do, remember that you read about it here first). Worse yet, someone might figure out how to exploit it two years down the track when software that's vulnerable to it has been widely deployed.

If this happens with a mainstream protocol like SSL or SSH, there'll be a CERT

Further reading

This article has only scratched the surface of what's involved in designing a security protocol. A good general reference on crypto and security issues, which (among other things) covers a range of security mechanisms, explains why many of them are insecure, and provides recommended fixes, is "Network Security: Private Communication in a Public World" by Charlie Kaufman, Radia Perlman, and Mike Speciner.

Bruce Schneier and Neils Ferguson's "Practical Cryptography" contains a step-by-step design of an SSL/SSH-like protocol, going through the multiple iterations of design needed to address each potential weakness and close off avenues of attack. "Lessons Learned in Implementing and Deploying Crypto Software", linked off my home page at *http://www.cs.auckland.ac.nz/~pgut001/*, looks at some of the problems that crop up when people use otherwise secure encryption building blocks in an insecure manner. In terms of IPsec VPN information, pickings are sparse: The RFCs contain almost no information on why they do things the way they do, and most IPsec books just reprint or paraphrase the RFCs. If you can handle the volume, there's a wealth of VPN design knowledge contained in the IPsec mailing list archives, *http://www.vpnc.org/ietf-ipsec/*, although you're in for a lot of reading if you want to cover all of it.

For background material, there's way too much to cover here (not helped by the fact that the details keep changing over time), but "The SIGMA Family of Key-Exchange Protocols" *http://www.ee.technion.ac.il/~hugo/sigma.html* is probably the most readable overview of the type of thing that IPsec's key management mechanism is doing.

advisory published and all the SSL/SSH vendors will rush out to fix things. The analysis that lead to this came too late for these particular protocols [10], however SSH isn't vulnerable to this specific problem and SSL implementors already know about it and provide workarounds, so it's a fairly safe example to use here.

However, if the same weakness is present in any other protocol, no-one will ever find out about it because the other protocols aren't mainstream enough for cryptographers to look at them. The fact that so many VPN apps are vulnerable to even the most trivial attacks shows just how dangerous this lack of outside security analysis can be.

Biological entities

When some new attack is discovered, it'll be tested against one of the big three (SSL, SSH, and IPsec), generally with some cross-pollination across the protocols. Applications will be immunized against the attack if they aren't already, and it'll live out its life in the proceedings of a crypto or security conference.

Like a biological organism, the big three will adapt and evolve to resist new attacks, although occasionally weaker strains like SSHv1 and SSLv2 will die out.

There was also an SSLv1, but it was broken in real time by members of the audience as it was being presented, which is why you've never heard of it before (it shared many characteristics with some of the protocols described earlier). Lesser-known protocols, on the other hand, could survive in isolation for years until the day they're exposed to the outside world, whereupon the first attack to come along could wipe them out completely.

This is not the sole reason why it's a good idea to go with a standard design. The problems found in implementations of undocumented homebrew designs mentioned in the previous section can't occur with a standard design because there's no ambiguity over what's a protocol issue and what's an implementation issue. If someone does get the implementation wrong, it won't be too hard to detect because their implementation won't interoperate with anyone else's.

For example the code for one of the VPN applications that I examined looked like it would send the first session key over the data channel in the clear because no session key had been established yet, although after about 45 minutes I gave up trying to dig the exact protocol details out of the mountain of uncommented spaghetti code. Such a flaw could never occur in one of the big three because (apart from the fact that they never directly exchange encryption keys) any implementation that did send a key in the clear would be quickly detected by the fact that nothing else would be able to talk to it.

In order to make the VPN selection/design process a bit easier, the two sidebars "Pre-built VPNs" and "Rolling your own" provide some general guidelines on choosing or implementing your own VPN software. ■

Table 1: Abbreviations

Abbreviation	Meaning
3DES	Triple DES
ACK	Acknowledge
AES	Advanced Encryption Standard
AH	Authentication Header
ARP	Address Resolution Protocol
CBC	Cipher Block Chaining
CERT	Computer Emergency Response Team
CFB	Ciphertext Feedback
CRC	Cyclic Redundancy Check
DES	Data Encryption Standard
DH	Diffie-Hellman
DHCP	Dynamic Host Configuration Protocol
DoS	Denial of Service
ECB	Electronic Codebook
EDI	Electronic Data Interchange
ESP	Encapsulating Security Payload
HMAC	Hashed Message Authentication Code
ICMP	Internet Control Message Protocol
IKE	Internet Key Exchange
IP	Internet Protocol
IPsec	Internet Protocol Security
IV	Initialisation Vector
MAC	Message Authentication Code
PGP	Pretty Good Privacy
PPTP	Point-to-Point Tunneling Protocol
QoS	Qualit of Service
RSA	Rivest Shamir Adleman
SHA1	Secure Hash Algorithm 1
SIGMA	Sign-and-MAC
SSH	Secure Shell
SWIFT	Society for Worldwide Interbank Funds Transfer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
VPN	Virtual Private Network
WEP	Wired Equivalent Protocol
XOR	Exklusive Or

INFO

[1] Performance of FreeS/WAN: *http://www.freeswan.org/freeswan_trees/freeswan-2.02/doc/performance.html*

[2] Bodo Möller, "TLS insecurity (attack on CBC)", posted to the IETF-TLS mailing list, September 2001. Also quoted in: *http://www.openssl.org/~bodo/tls-cbc.txt*

[3] Wei Dai, "an attack against SSH2 protocol", posting to the *sci.crypt* newsgroup, February 2002

[4] Phil Rogaway, "Problems with Proposed IP Cryptography", April 1995: *http://www.cs.ucdavis.edu/~rogaway/papers/draft-rogaway-ipsec-comments-00.txt*

[5] Ariel Futoransky, E. Kargieman, and Ariel M. Pacetti, "An attack on CRC-32 integrity checks of encrypted channels using CBC and CFB modes", CORE SDI S.A, 1998

[6] Steven M. Bellovin, "Problem Areas for the IP Security Protocols", Proceedings of the 1996 Usenix Security Symposium, August 1996, p205

[7] Stephen Kent and Randall Atkinson, "RFC 2406: IP Encapsulating Security Payload (ESP)", November 1998

[8] Nikita Borisov, Ian Goldberg, and David Wagner, "Intercepting Mobile Communications: The Insecurity of 802.11", Proceedings of the 7th Annual International Conference on Mobile Computing and Networking, 2001, p180

[9] Bruce Schneier and Mudge, "Cryptanalysis of Microsoft's Point-to-Point Tunneling Protocol (PPTP)", Proceedings of the 5th ACM Conference on Communications and Computer Security, November 1998, p132: *http://www.schneier.com/paper-pptp.html*

[10] Hugo Krawczyk, "The order of encryption and authentication for protecting communications (Or: how secure is SSL?)", Proceedings of Crypto'01, Springer-Verlag Lecture Notes in Computer Science No.2139, August 2001, p310

[11] Karl-Heinz Haag and Achim Leitner, series on OpenSSH, Linux Magazine #24, p50; #25, p48; #49, p52

[12] Ralf Spenneberg, "Virtual Private Networks with Linux 2.4 and FreeS/WAN 2.01", Linux Magazine #36, p52