Attacking the IPsec Standards in Encryption-only Configurations

Jean Paul Degabriele[1] and Kenneth G. Paterson[2] *

[1] Hewlett-Packard Laboratories, Bristol
Filton Road, Stoke Gifford, Bristol BS34 8QZ, UK.
jeanpaul.degabriele@gmail.com
[2] Information Security Group,
Royal Holloway University of London,
Egham, Surrey TW20 0EX, UK.
Kenny.Paterson@rhul.ac.uk

Abstract. At Eurocrypt 2006, Paterson and Yau demonstrated how flaws in the Linux implementation of IPsec could be exploited to break encryption-only configurations of ESP, the IPsec encryption protocol. Their work highlighted the dangers of not using authenticated encryption in fielded systems, but did not constitute an attack on the actual IPsec standards themselves; in fact, the attacks of Paterson and Yau should be prevented by any standards-compliant IPsec implementation. In contrast, this paper describes new attacks which break *any* RFC-compliant implementation of IPsec making use of encryption-only ESP. The new attacks are both efficient and realistic: they are ciphertext-only and need only the capability to eavesdrop on ESP-encrypted traffic and to inject traffic into the network. The paper also reports our experiences in applying the attacks to a variety of implementations of IPsec, and reflects on what these experiences tell us about how security standards should be written so as to simplify the task of software developers.

Keywords: IPsec, integrity, encryption, ESP, standard.

1 Introduction

The need to use integrity protection along with encryption in order to prevent active attacks is well-understood in the theoretical cryptography community. Moreover, there are plenty of high-profile examples where lack of strong integrity checks (or their inappropriate application) has allowed attacks to succeed [2–5, 21]. Yet "encryption-only" configurations are still allowed by the IPsec standards [10, 12], and may still be in quite common use in spite of Bellovin's classic attacks against them [3]. Instead of choosing to eliminate configurations known to have weaknesses altogether, the IPsec RFCs have included progressively stronger warnings about encryption-only configurations, together with a recommendation that implementations should check the correctness of encryption padding when performing in-bound processing. The latter check, if implemented properly, is sufficient to prevent certain specific attacks of Bellovin from working.

However, the placement of warnings to implementors in the standards does not prevent implementors from making weak configurations available to end-users. Indeed, support for encryption-only configurations was mandatory until the most recent generation of IPsec standards. Nor do warnings in RFCs prevent end-users from selecting such weak configurations. In case there is any doubt about this, one only needs to consult the on-line manual pages for market-leading VPN products, where one can find statements such as:

> *If you require data confidentiality only in your IPSec tunnel implementation, you should use ESP without authentication. By doing so, you gain some performance speed but lose the authentication service.* [6]

* This author's research supported in part by Hewlett-Packard Laboratories, BT Laboratories, and the EPSRC through the Industrial CASE scheme, and in part by the European Commission under contract IST-2002-507932 (ECRYPT).

Moreover, as we shall see, despite the recommendations concerning padding checks in the RFCs, many open-source implementations still fail to check encryption padding correctly, even though this renders them vulnerable to Bellovin's 10-year old attacks.

Paterson and Yau [17] aimed to show that encryption-only configurations of IPsec are fatally flawed. They described realistic attacks against encryption-only, tunnel mode IPsec and showed that the attacks worked against the Linux kernel implementation of IPsec. However, as was made clear in [17], these attacks do not apply to an implementation of IPsec that is faithful to the relevant IPsec standards. In particular, the attacks do not work if certain post-processing policy checks specified in the IPsec architectural RFCs [9, 11] are actually carried out as they should be. The underlying reason for this is that the modified IP datagrams used in the attacks would not pass the policy checks and would simply be dropped by IPsec. The Linux implementation of IPsec does not carry out these checks.

Thus [17] is only partially successful in highlighting the dangers of encryption-only IPsec. Indeed, the work of [17] can be regarded as providing attacks against a specific and incomplete implementation of the IPsec standards, rather than attacks against the standards *per se*. This point of view brings with it the danger that implementors and users of IPsec might go on to dismiss the work of [17], assuming instead that a standards-compliant implementation of IPsec would still be fit-for-purpose in encryption-only configurations. After all, bad security implementations are commonplace; bad security standards hopefully less so. This paper sets out to show that the broad community of IPsec users should not be lulled into such a false sense of security, by providing attacks against encryption-only IPsec as specified in the IPsec RFCs themselves.

Our new attacks are both realistic and efficient, and we have studied their applicability to a variety of open-source IPsec implementations. In particular, we are able to use them to break the OpenSolaris IPsec implementation. In contrast to the chosen-plaintext requirements of the attacks of Bellovin [3], our new attacks are ciphertext-only. Our main attacks are applicable *only* if an IPsec implementation performs a full padding check, in accordance with the recommendations of the IPsec RFCs. Thus, our attacks work where Bellovin's are prevented, and vice-versa. We also sketch variants of our ciphertext-only attacks that work when less rigorous padding checks are implemented. Taken as a whole, the various attacks show that IPsec in encryption-only configurations is vulnerable, whether or not implementors follow the RFCs and carry out proper padding checks.

1.1 Our Contributions

We provide new attacks against encryption-only IPsec as specified in the RFCs. For concreteness, we work with the latest generation of IPsec standards, RFCs 4301-4309, but our attacks also apply for the second generation of IPsec standards, RFCs 2401-2411. Our attacks can extract the complete contents of IPsec-encrypted datagrams and remove a significant barrier to the operation of the best attacks in [17]: they do not require the attacker to be able to monitor all the traffic emanating from a host performing IPsec, but instead just the traffic flowing in two directions in an IPsec tunnel. For simplicity, we focus in this paper on tunnel mode IPsec, but we also sketch how to apply our ideas to transport mode. We also show that variants of our attacks can be used to defeat the two traffic flow confidentiality mechanisms that have been included in the newest versions of the IPsec standards (when configured in encryption-only mode). These attacks allow an attacker to efficiently separate ciphertext blocks of real interest from dummy traffic. We report on our experiences in applying the new attacks to various IPsec implementations, in particular, our success in breaking the OpenSolaris implementation.

Our new attacks involve a combination and significant extension of ideas from [17] and the padding oracle attacks of Vaudenay [20]. The main idea we borrow from [17] is to manipulate selected header fields of inner datagrams protected using ESP, the encryption protocol of IPsec, so that they consistently lead to the creation of ICMP messages. However, because of the order of processing in IPsec and IP, an ICMP message will only be generated if the padding and certain other fields in the ESP trailer that follows the inner datagram are correctly formatted. Here we are assuming that an implementor has followed the advice of the relevant RFCs [10, 12] that padding

checks SHOULD[3] be carried out. Thus the absence or presence of the ICMP messages can be used to build an *ESP trailer* oracle, an extension of Vaudenay's padding oracle concept. To complicate matters, because of constraints on our attack imposed by the need to satisfy policy checks specified in the IPsec RFCs, these ICMP messages will only be available to the attacker in encrypted form. However, they have stereotypical lengths which allow them to be detected nevertheless. The single bit of information revealed by our ESP trailer oracle can then be leveraged to extract the entire contents of IPsec-encrypted datagrams.

We note that Vaudenay [20] has already sketched how a padding oracle attack against IPsec might operate *if* a suitable padding oracle were available. However, the attack sketched in [20] will not work as described, for reasons explained in more detail in Section 3. Indeed, a degree of ingenuity is needed to turn this sketch into a working attack: we need to find a means of building the oracle, and we need to overcome a variety of technical obstacles which interfere with the operation of Vaudenay's basic attack. A second contribution of this paper, then, is to flesh out the IPsec attack sketched in [20], in a similar way as Canvel *et al.* [5] did when turning the padding oracle attack against SSL/TLS described in [20] into an attack against an actual implementation of SSL/TLS. This paper, like [5], demonstrates the real-world applicability of padding oracle attacks, showing that they are not just a theoretical nicety.

A further theme of this paper is to illustrate the complexities that arise in the real-world cryptanalysis of secure protocols. As we shall see, significant effort must be invested to turn attacks that work "on paper" into attacks that would actually work in practice against systems compliant with relevant standards. However, as we shall also discover, whether or not a particular attack works against a particular implementation depends in a complex way on the extent to which the implementor has deviated from the specification provided by the standards, or has chosen to "interpolate" details missing from the standards.

1.2 Overview of Paper

In the next section we provide additional background on IPsec and related topics. We focus on how padding and CBC mode encryption are carried out in ESP in tunnel mode. Section 3 discusses padding oracle attacks [20] and explains why, in their basic form, they are unlikely to succeed against IPsec implementations. Section 4 presents a simple chosen-plaintext attack on encryption-only ESP, which illustrates the main ideas in the ciphertext-only attacks to follow. As with [17], our attacks are somewhat different in character, depending on whether the underlying block cipher used by ESP (the IPsec encryption protocol) has 64-bit or 128-bit blocks. An attack well-suited to the 64-bit case is discussed in Section 5 and an attack for the 128-bit case is given in Section 6. We briefly discuss how to defeat the traffic flow confidentiality mechanisms in IPsec in Section 7. Section 8 explains how our ideas can be adapted to attack transport mode configurations of IPsec. Our experiences in attacking implementations of IPsec using our new ideas are reported in Section 9. We conclude in Section 10.

2 Background

2.1 IPsec

IPsec, as defined in RFCs 2401–2412 and 4301–4309, provides security at the IP layer. The interested reader is invited to consult [7, 8] for accessible introductions to IPsec. We will make free use of IP and IPsec terminology in what follows.

We recall that the ESP protocol, as defined in [10, 12] principally provides a confidentiality service, and usually makes use of a block cipher algorithm operating in CBC mode. We assume

[3] "SHOULD" is one of the key words used in RFCs to indicate requirement levels. It's meaning is defined in RFC 2119 as follows: *This word, or the adjective "RECOMMENDED", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.*

throughout the remainder of this paper that CBC mode is in use. The algorithms, mode, keys, IVs and other associated data to be used in encrypting network traffic are stored in an object called a Security Association (SA); the selection of SAs applied to a particular datagram is determined by matching fields in IP datagrams to IPsec security policies.

In tunnel mode ESP, entire IP datagrams, called inner datagrams, are protected; each inner IP datagram is encrypted and forms part of the payload of an outer IP datagram. In transport mode ESP, only the payload of a datagram is encrypted, and the original datagram header is modified to include IPsec-specific information. ESP may also be configured to provide integrity protection through the application of a MAC algorithm, or this service can be provided by a separate IPsec protocol, AH. In the second generation of IPsec standards, including support for encryption-only operation of ESP was a "MUST" [10]. This requirement has been weakened to a "MAY" in the third generation of standards [12]. Both generations do warn against the use of encryption-only ESP without some additional integrity protection (being provided either by ESP itself, by combining ESP with AH, or by an upper-layer protocol). Paterson and Yau [17] provide further background on the history of integrity protection in ESP.

2.2 Padding and CBC Mode Encryption in Tunnel Mode ESP

Our attacks depend crucially on how padding and CBC mode encryption (and the reverse operations of decryption and depadding) are performed by IPsec. The variant of CBC mode that is used by ESP in tunnel mode operates as follows. The original (inner) datagram that is to be protected is treated as a sequence of bytes. This sequence is padded with a particular pattern of bytes and then a Pad Length (PL) byte and a Next Header (NH) byte are appended. Each different encryption algorithm for use with ESP is specified in a separate RFC, and each such RFC may specify its own padding method. However none of the relevant RFCs appears to do so, and so the default padding method specified in [10, 12] is universally used in practice. This method adds bytes so that:

1. The total number of bytes present (including the PL and NH byte) is aligned with a block boundary; and
2. The added pattern of padding bytes is either a null string or t bytes of the form $1, 2, \ldots, t$ for some t with $1 \leq t \leq 255$.

It is permissible for the padding to be of variable length, though this option seems to be rarely exercised in practice – usually the minimum amount of padding consistent with the above rules is added (one notable exception is the KAME implementation.[4]) According to [12, Section 2.7] it is permissable in tunnel mode to precede this padding with an arbitrary amount of Traffic Flow Confidentiality (TFC) padding too. This is intended to aid in preventing traffic analysis by disguising the true length of the inner datagram. Our attacks have no problem dealing with either type of extended padding. Note that even minimal padding may extend over multiple blocks. For example, if the block size is 64 bits (8 bytes) and the IP datagram ends at byte 7 in a block, then the padding pattern would be:

$$1, 2, 3, 4, 5, 6, 7$$

followed by the PL byte (in this case, equal to 7) and the NH byte. Here, the first byte with value 1 would complete a block, and the 6 remaining padding bytes together with the PL and NH bytes would appear in an extra block.

The NH byte is present in order that the decrypting IPsec entity can know to which protocol implementation it should pass the bytes that precede the padding. In tunnel mode, this value should be 4, indicating IP-in-IP encapsulation and that the bytes are indeed an IP datagram. The latest ESP RFC [12] specifies that an NH byte value of 59 indicates a dummy packet. Such datagrams are to be ignored upon decryption and so can be used along with TFC padding to build a traffic flow confidentiality service.

[4] `http://www.kame.net`

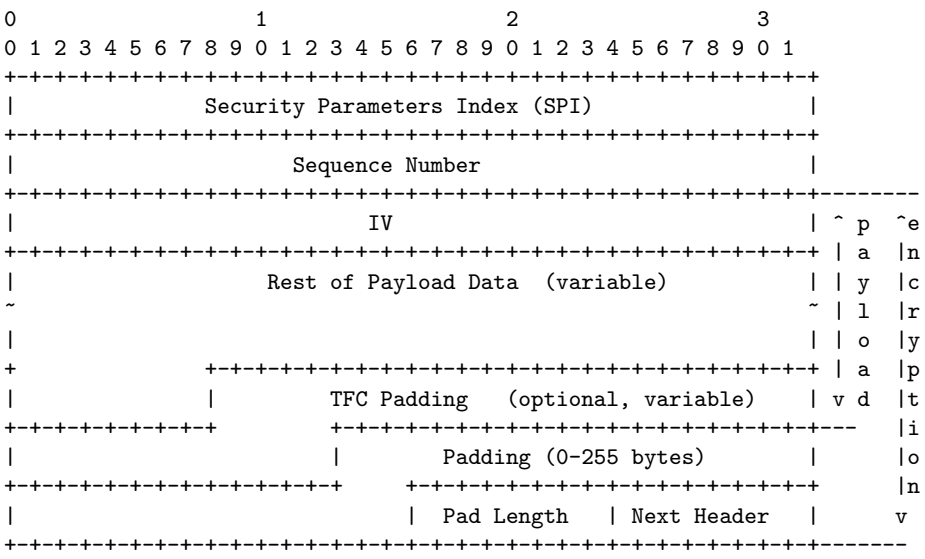

Fig. 1. Structure of ESP protected-datagram (adapted from RFC 4303 [12] for CBC mode without integrity protection and to show encryption scope).

After padding, the data is encrypted using CBC mode. Let us assume that the byte sequence after padding consists of q blocks, each of n bits (where $n = 64$ for triple-DES and $n = 128$ for AES, for example). We denote these blocks by $P_1, P_2, \ldots, P_q$. We use K to denote the key used for the block cipher algorithm and $e_K(\cdot)$ ($d_K(\cdot)$) to denote encryption (decryption) of blocks using key K. An n-bit initialization vector, denoted IV, is selected at random. Then ciphertext blocks are generated according to the equations:

$$C_0 = IV, \quad C_i = e_K(C_{i-1} \oplus P_i), \quad (1 \le i \le q).$$

The encrypted portion of the outer datagram is then defined to be the sequence of $q + 1$ blocks $C_0, C_1, \ldots, C_q$. The basic format of an ESP-protected datagram is shown in Figure 1.

2.3 CBC Mode Decryption and Depadding in Tunnel Mode ESP

At the entity performing IPsec decryption (which is also in possession of the key K), the payload of the outer datagram can be recovered using the equations:

$$P_i = C_{i-1} \oplus d_K(C_i), \quad (1 \le i \le q).$$

Any padding along with the PL and NH bytes can then be stripped off, revealing the original inner datagram. Section 2.4 of both the ESP RFCs [10, 12] states that "*the receiver SHOULD inspect the padding field*", because certain cut-and-paste attacks are prevented if "*the receiver checks the padding values upon decryption*". This is apparently a reference to the chosen-plaintext attack of Bellovin [3] which can extract 1 byte per block from ciphertexts of special lengths and which depends for its success on the padding *not* being properly checked. What kind of inspection is required is not made clear. Moreover, it is not made explicit in the ESP RFCs what action should be taken in the event that the padding does not have the correct format. It is not explicitly specified that the datagram should be dropped, nor that the event is auditable.

However, padding checks are only effective as a countermeasure against Bellovin's attack if the padding is checked strictly and the datagram dropped if the padding is not fully conforming. For

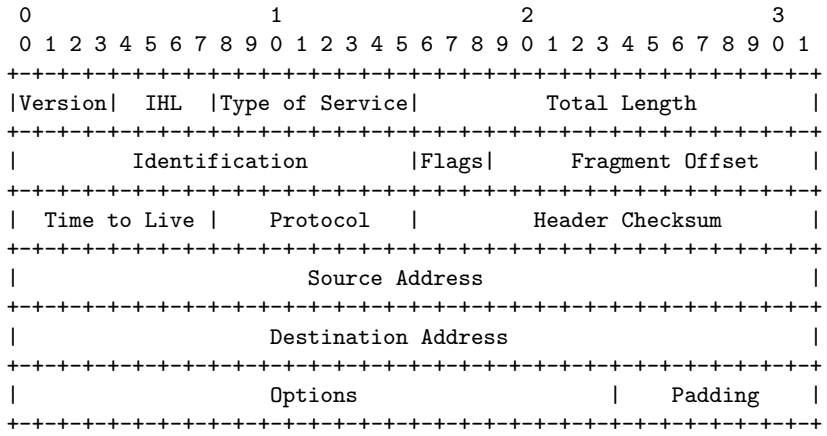

Fig. 2. Structure of IP header according to RFC 791, [14].

this reason, we assume that a compliant implementation *does* perform a strict check, checking that the padding conforms exactly to what is expected given the value of the PL field, and dropping the inner datagram if the check fails.

We will see in Section 9 that, in the absence of a detailed specification in the RFCs, implementers have taken a variety of approaches to handling depadding. As we shall also see, appropriate handling of this function is security-critical. It is implicit in the text of the ESP RFC [12, p.30] that the removal of any TFC padding is the responsibility of the upper layer protocol specified in the NH byte.

The ESP RFCs next specify that the NH byte should be examined, and if the value is 59, then the packet is discarded without further processing. At this point, the receiver is able to reconstruct the original inner datagram that was encrypted. The exact process for doing so is specified in the IPsec architectural RFCs [9, 11]. These further mandate that implementations should check that the cryptographic processing performed to recover the inner datagram does in fact match that specified in local IPsec policies. If the check fails, the datagram should be dropped [11]. If the check passes, then (in tunnel mode) the inner datagram is passed on to the IP implementation for further processing.

Note that when ESP is employed without integrity protection, the sequence number in the ESP header is not checked by the recipient.

2.4 Transport Mode ESP

Much of the discussion in the previous two-subsections applies to transport mode ESP as well. Major differences include the inclusion of an upper layer protocol number in the NH byte to indicate the content of the protected payload (for example, 6 for TCP, 11 for UDP, etc), and the fact that the payload resulting from depadding and decryption is passed to an upper layer implementation for further processing, rather than to IP.

2.5 IP Datagram Headers

Our attacks depend in a detailed way on the structure of the headers of IP datagrams, on the order in which the fields of inner IP datagram headers are processed by an implementation of IP after IPsec processing has completed, and on the way in which ICMP errors are generated in the event of certain errors in processing. Figure 2 shows the structure of an IP datagram header.

In essence, the processing is as follows in a typical implementation of IP (where, for simplicity of presentation, we ignore any fragmentation processing).

Basic checks are performed on the Version field and IHL field. The IHL (Internet Header Length) field is 4 bits long and has a value between 5 and 15. This field indicates the length of the header in 32-bit words. The typical value is 5, indicating that the header length is 20 bytes and no additional options bytes are present. If the IHL value is greater than 5, then additional options bytes are assumed to be present after the main header, in the Options field.

The next action is to check the Header Checksum field. This 2-byte field is initially formed by interpreting the header (including the Options field if present) as a sequence of 16-bit words, summing them using 1's complement arithmetic, and then taking the 1's complement of the result as the Header Checksum value. During this computation, the Header Checksum bytes are set to zero. As a consequence, the 1's complement sum of the 16-bit header words (now including the computed Header Checksum field) should equal zero. If this check fails, the datagram is simply dropped.

After this, length checking is carried out. A typical implementation like Linux will check that the number of bytes in the datagram is at least as many as are indicated in the Total Length field, and at least the minimum size of an IP header (20 bytes). The datagram will then be trimmed to the length indicated in the Total Length field. The datagram is dropped if these checks fail. We refer to this style of length checking as *relaxed*. *Strict* length checking, in which an IP implementation ensures that the number of bytes in the datagram exactly matches the number indicated in the Total Length field and rejects the datagram if not, could be carried out instead. However, a lower layer implementation cannot necessarily know how to trim its payloads to present the IP implementation with a datagram of exactly the right size. Moreover, for the TFC padding mechanism to work as implicitly defined in [12], we must assume that IP implementations perform relaxed checking and then discard any "redundant" bytes beyond the length indicated in the Total Length field. This is consistent with the IP specification [14], which does not explicitly specify any kind of length checking, but does state that "an implementation should be liberal in its receiving behaviour". In the remainder of this paper, we will assume that relaxed length checking is performed (but we will indicate how our attacks can be modified to handle strict checking).

Next, options processing is carried out if the IHL field indicates that options are present. The Options field has a strict format. If the format is not followed, then the datagram is dropped and IP generates an ICMP "parameter problem" message which is routed to the host indicated in the Source Address field.

Then a routing decision is made: either the datagram is delivered locally or it is forwarded to another host (if this host is configured for routing). In the former case the Protocol field is used to determine the upper layer protocol to which the datagram payload should be passed. If the field contains a value corresponding to a protocol that is not supported at the host, then IP should drop the datagram and generate an ICMP "protocol unreachable" message to be sent back to the host indicted in the source address. Slightly more than half of the 256 possible values of the Protocol field are already allocated to specific upper layer protocols (such as TCP and UDP), but a typical host may only support 5-10 different protocols. In the latter case, where the datagram is to be forwarded, the Time to Live (TTL) field is then checked. The TTL field is used to track the "age" of datagrams. It is set to an initial value by the sending host, typically 64 or 128, and decreased by 1 by each intermediate router visited by the datagram. If the TTL field reaches zero, the datagram is dropped and an ICMP "Time Exceeded" error message indicating this event is sent back to the host at the source address in the datagram. Otherwise, the datagram is forwarded.

We will make use of the IHL and Protocol fields in our attacks, noting their propensity to provoke the generation of ICMP messages in the event of their containing certain values. We will also exploit the Time to Live (TTL) and Identification fields in some of our attacks. The Identification field carries a 16-bit value that is used to track the fragments making up a datagram in the event that the datagram is fragmented. Typically, an implementation of IP fills this field with a counter that is incremented for each new datagram.

2.6 ICMP and IPsec

ICMP was originally specified in [19], and revised for IPv4 routers in [1]. We have already indicated above a number of circumstances under which ICMP messages will be generated during IP processing. The particular format and length of the payload of these messages will depend both on the error type and on the implementation of ICMP.

The IPsec architectural RFCs [9, 11] contain complex prescriptions for how IPsec should handle ICMP messages. RFC 4301 [11] distinguishes between error and non-error messages, with the latter type being catered for by specific security policies and SAs. ICMP error messages are divided into two types: messages directed to the IPsec implementation itself, and transit messages. All the ICMP error messages arising in our attacks will be of the transit type. The manner in which these are to be handled by IPsec processing is described in [11, Section 6.2]. The highlights are as follows. In a first case, either an SA already exists that accommodates the ICMP message, or IPsec security policy allows one to be created. In this case, out-bound and in-bound IPsec processing proceed as normal. In a second case, no SA exists and IPsec security policy does not allow one to be created. In this situation, IPsec out-bound processing must use the SA that would be used for return traffic corresponding to the traffic that generated the ICMP message in the first place. This SA can be identified by examining the ICMP payload, as this carries the header and part of the payload of the error-provoking IP datagram. Thus, even if no policy applicable to ICMP traffic is extant, the ICMP traffic will still be sent protected by IPsec. However, it is possible that no SA exists for the return traffic, in which case the RFC specifies that the message should be dropped and an auditable event occurs. In-bound processing in the second case involves an extra check to ensure that the SA applied is consistent with the fields in the IP datagram carried in the ICMP payload.

The key point to remember from this discussion is that, while IPsec processing of ICMP messages can be complicated, our attacks only involve transit ICMP error messages. In the IPsec configurations we attack, these will not be blocked by IPsec and will travel in encrypted form across the network. They will be detectable through their characteristic (though implementation-dependent) lengths, or via related techniques that are discussed in Section 4.

2.7 Bit Flipping in CBC Mode

We recall the following well-known property of CBC mode. Suppose an attacker captures a CBC mode ciphertext $C_0, C_1, \ldots, C_q$, then flips (inverts) a specific bit j in C_{i-1} and injects the modified ciphertext into the network. Then the attacker induces a bit flip in position j in the plaintext block P_i as seen by the decrypting party. This tends to randomize block P_{i-1}, but if the modification is made in C_0 (equal to IV), then no damage to plaintext blocks will result.

2.8 Correcting Checksums after Bit Flipping

In our attacks, we will be modifying certain bits in the headers of inner datagrams. Any such modifications will require further compensation to be made elsewhere in the header so that the Header Checksum is still correct – otherwise the inner datagram will be silently dropped. In [17], two basic techniques were used to achieve this: firstly, randomisation of the Header Checksum field (possibly along with other fields) by making changes to non-IV ciphertext blocks, and, secondly, bit flipping in positions in the IV corresponding to the checksum itself. The first technique tends to lead to changes to source and/or destination addresses in the inner datagram, which results in datagrams being dropped by IPsec policy checks performed during in-bound processing. The second technique had two manifestations: trying all 2^{16} 16-bit values, and, when only small numbers of bit flips to the headers were involved, using sequences of correcting masks. In the 64-bit case, this second technique does not work, because the checksum is located in the second plaintext block. Because of these limitations, we will use a modified approach to ensure that correct Header Checksums are maintained. The approach is based on ideas in [15].

We recall that the condition for the Header Checksum to be correct is that the 1's complement sum of the 16-bit header words (including the Header Checksum field) should equal zero. This means that, from the point of view of further modifying an inner header to compensate for earlier modifications to that header, there is no need to stipulate (as in [17]) that it is the Header Checksum field itself that must be manipulated: any 16-bit header word can be used for this purpose. However, not every 16-bit header word is equally well-suited to this job, since they contain fields whose values may adversely affect subsequent IP processing. But recall that the 16-bit Identification field, located in bytes 5 and 6 of the datagram header, can quite legally take on any 16-bit value without affecting the processing of unfragmented datagrams. Because of its location in the header, this field can be modified by bit-flipping in bytes 5 and 6 of the IV, both in the 64-bit and 128-bit cases. These properties make the Identification field an ideal 16-bit header word for working with to ensure we obtain correct checksums.

Finally, we note that we can use the Identification field in two ways. If many bit changes have been made to the header, then we can systematically vary the value of the Identification field over all possible 16-bit values, using a 16-bit counter to modify bytes 5 and 6 of the IV. An average of 2^{15} and at most 2^{16} trials will be needed to produce a header with a valid checksum value. If a small number of bit changes have been made to the header, then we can use the tables of masks from [17] to vary the value of the Identification field. Although these masks were designed for modifying the Header Checksum field itself, it is clear from the discussion above that they will work just as well when applied to the Identification field. So for example, suppose a single bit change is made to the header. Then we know that using the masks from an appropriate table T_i in [17] to modify bytes 5 and 6 of the IV, we will need to try at most 17 masks, and on average 2 masks, to produce a header with a valid checksum value.

3 Padding Oracle Attacks

We provide a brief introduction to padding oracle attacks, as introduced by Vaudenay in [20]. In such an attack, the attacker has access to a padding oracle that, on receipt of a ciphertext, outputs a single bit indicating whether or not the underlying plaintext is correctly padded.

It is shown in [20] that, for CBC mode encryption and a variety of padding methods, repeated use of a padding oracle can be used to build a decryption oracle. We sketch how this can be done for the default padding method used by ESP, assuming the existence of a suitable padding oracle. To simplify the presentation, for now we assume that the NH byte is not present, so that the PL byte appears in the rightmost byte of a block.

Suppose the attacker wishes to decrypt a target ciphertext block C_i, $i \geq 1$, from a CBC mode ciphertext consisting of blocks $C_0, C_1, \ldots, C_q$. Let us assume that each block C_j has t bytes, labeled $C_{j,0}, C_{j,1}, \ldots, C_{j,t-1}$ from left to right. We label the unknown bytes of $d_K(C_i)$ as $(d_K(C_i))_0, (d_K(C_i))_1, \ldots, (d_K(C_i))_{t-1}$, and the unknown bytes of P_i as $P_{i,0}, P_{i,1}, \ldots, P_{i,t-1}$. The attacker creates a two-block ciphertext of the form R, C_i and submits this to the oracle. Here R with bytes $R_0, R_1, \ldots, R_{t-1}$ is a random block. If the oracle indicates the padding is correct, then the rightmost byte of $R \oplus d_K(C_i)$ is most probably equal to 0, a PL byte indicating that no padding bytes are present. (The next most probable valid combination of padding and PL byte is $1, 1$. This is 256 times less likely to occur and can be excluded using an extra oracle query.) Then the attacker can easily calculate $(d_K(C_i))_{t-1}$ and hence $P_{i,t-1}$. If the oracle indicates incorrect padding, the attacker can try again with a different value for R_{t-1}. It is now easy to see how an attacker who systematically varies byte R_{t-1} and makes padding oracle queries can extract the rightmost byte of P_i using on average 128 and at most 256 queries.

This idea can be extended to extract every byte of P_i. Suppose the rightmost byte has been extracted, so the attacker has learned $(d_K(C_i))_{t-1}$. To get byte $P_{i,t-2}$, we can fix R_{t-1} so that $R_{t-1} \oplus (d_K(C_i))_{t-1} = 1$ and then vary R_{t-2} until a correct padding is indicated by the oracle. When this happens, we know that the padding and PL byte must be of the form $1, 1$. From this, byte $P_{i,t-2}$ can be deduced. The decryption process continues in this way, with the padding pattern being increased in length by one at each step. Recovering each plaintext byte requires on average 128 queries to the padding oracle.

3.1 Applicability to IPsec

Unfortunately, a padding oracle attack, as described above, will not work against an implementation of IPsec. First,we have not identified whether a padding oracle really exists, or a mechanism by which the output of the oracle becomes known to the adversary. Our analysis above of the relevant RFCs suggests that strict padding checks should be carried out, but a failure is simply logged and the offending datagram dropped. In [20], it is remarked that, in the event of incorrect padding in ESP,"*It is reasonable to assume that the lack of activity of the receiver ..., or the activity of the auditor, can be converted into one bit of information*". Detecting the activity of the auditor would seem to need the attacker to have access to the relevant log file, or to be able to measure the length of that file. It is not immediately clear how a *lack* of activity because of a padding failure can be distinguished from lack of activity that might arise for a multitude of other reasons. This should be contrasted with the situation with SSL/TLS, where a padding failure leads to the generation of an error message, an active event that can be detected (although it also results in the tear-down of the SSL connection) [20, 5].

Furthermore, the attack as described does not take into account the presence of the NH byte and its effect on subsequent processing of the decrypted-and-depadded data. The attack works by placing the target ciphertext block as the last block in a ciphertext, after a random block. This implies that the NH byte is effectively randomized in the attack as described. By changing R, the NH byte can be varied, but it is not clear at the outset what effect modifying this byte has. In fact, if post-processing policy checks are carried out and tunnel mode is in use, then the decrypted-and-depadded data will always be discarded unless the NH byte is equal to 4: any other value would indicate an upper layer protocol that was not IP, and hence would indicate that transport mode ESP was being used, leading to a policy violation. Thus, in this situation, correctness of padding alone will not be enough to ensure that data is not simply discarded; we will also need the NH byte to take on a particular value. In the analysis that follows, we assume that, in tunnel mode processing, the decrypted-and-depadded data will always be discarded unless the NH byte is equal to 4.

Even supposing the NH byte could be arranged to equal 4 so as to indicate tunnel mode, there is the question of how the decrypted-and-depadded data, now interpreted as an IP datagram, is subsequently processed. The attack as described above uses ciphertexts of the form R, C_i where C_i is the target block. Data produced after decryption and depadding is very unlikely to correspond to a valid IP datagram. In this case, subsequent processing by IP is almost certain to lead to a swift packet drop, resulting in a lack of activity by the receiver that seems likely to be indistinguishable from inactivity due to a padding failure. Thus we need to be much more careful in the way in which we compose ciphertexts if a padding oracle attack is to be successful.

4 A Tutorial Chosen-Plaintext Attack

In this section, we describe a chosen-plaintext attack that illustrates the main principles of the ciphertext-only attacks in the sections to follow. For concreteness, we suppose the block size of the block cipher used in ESP to be 64 bits. It is trivial to adapt the attack in this section to the 128-bit case. As with all of the attacks in this paper, we make the following assumptions:

1. Encryption-only ESP is used in tunnel mode between a pair of security gateways G_A and G_B; these gateways provide security for pairs of communicating hosts H_A and H_B located behind the gateways as illustrated in Figure 3. This is a typical VPN configuration of IPsec.
2. The key K used to perform ESP encryption for IP traffic flowing from G_A to G_B is fixed during the attack;
3. The attacker can monitor and capture ESP-protected datagrams that flow between the two gateways; and
4. The attacker can inject modified datagrams into the network between G_A and G_B.

Notice that the monitoring requirement here is more realistic than in [17], where the ability to observe all traffic emanating from a gateway was needed.

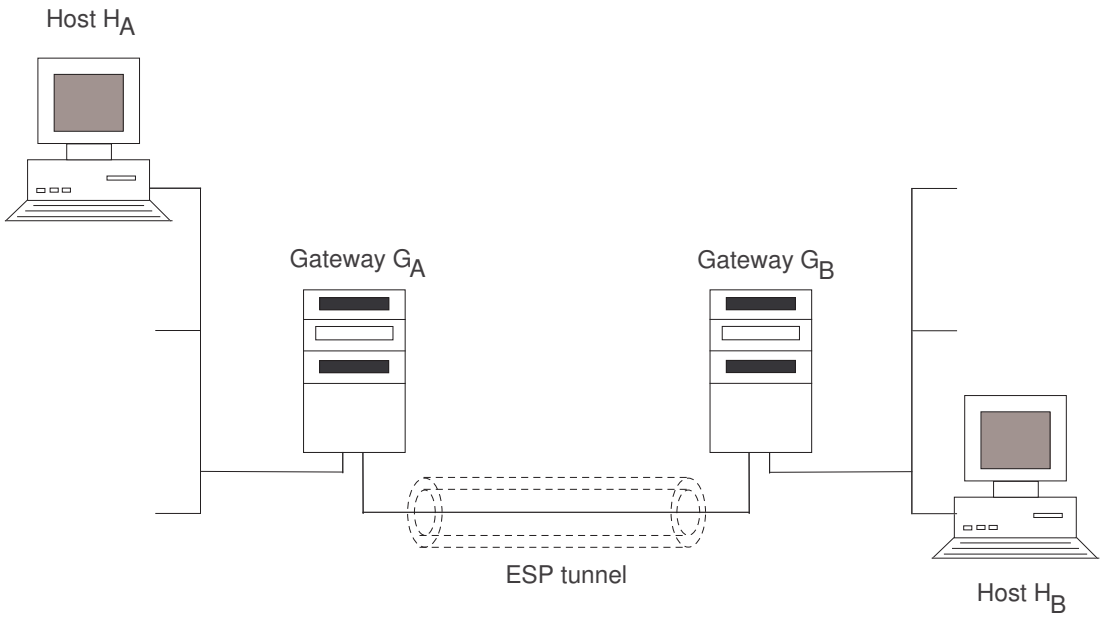

Fig. 3. Network setup.

We assume that appropriate SAs are in place to protect traffic from G_A to G_B and vice-versa. As discussed in Section 2.6, additional ICMP-specific SAs might be used to protect ICMP messages flowing from G_A to G_B and back again. Our attack works irrespective of whether the same SAs are used to protect both ICMP and non-ICMP traffic.

We suppose the attacker has collected a target ciphertext $C = C_0, C_1, \ldots, C_q$ from the payload of an outer IP datagram that was directed towards G_B. This could represent traffic flowing from H_A to H_B, for example. The attacker's goal is to discover the plaintext corresponding to C.

4.1 Preparation

We assume the attacker has obtained the ciphertexts C^j corresponding to a set of 7 chosen plaintexts P^j $(0 \leq j \leq 6)$. Here, P^j is selected to contain an inner IP datagram with source address set to that of H_A, destination address set to that of H_B and a TTL field set to 1. We further assume that P^j contains exactly $j + 12$ bytes of payload data after the IP header. These bytes need not be in the format of any particular upper layer protocol. Thus P^j begins with 20 bytes of IP header, followed by $j + 12$ bytes of payload data, a total of $j + 32$ bytes. Plaintext P^j also contains padding and the PL and NH bytes. We see that P^j has j bytes of payload in the final block, so, following the default padding rule, this will be followed by $8 - j - 2$ bytes of padding, the PL byte, and finally the NH byte (equal to 4). For example, P^6 ends with the byte pattern $0, 4$, while P^0 ends with the byte pattern $1, 2, 3, 4, 5, 6, 6, 4$ which fills the entire last block. Thus we see that P^j has 5 blocks, and the corresponding ciphertext C^j has 6 blocks (including the IV).

If C^j is injected into the network as the payload of an outer datagram with destination address equal to that of G_B, then the corresponding inner datagram will be recovered at G_B and passed to the IP implementation at G_B. This will decrement the TTL field of the inner datagram, producing a TTL value of zero. The IP implementation at G_B will then produce an ICMP error message (of type 11 and code 0) indicating a "time to live exceeded" event. Assuming the IP implementation follows [19] and includes the header and first 64 bits of payload of the original datagram, the total length of the ICMP message (including the header of the IP datagram in which it is carried) will

be 56 bytes. The error message will have the host H_A as its destination, and so will become an ESP-encrypted inner datagram at G_B. Using the padding rule, an extra block containing padding and the NH byte will be added, and so the resulting ciphertext will have 9 blocks (including the IV). Thus, an outer datagram carrying a 9-block ciphertext will be seen traveling in the reverse direction on the tunnel shortly after the injection of C^j. In principle, this ICMP message can be detected by the attacker (based on its length) even though it is carried inside a datagram that has been encrypted by IPsec.

If TFC padding or variable length encryption padding is in use on the tunnel, then the relevant ICMP message may be harder to separate from other encrypted network traffic. In this case, we can look for correlations in time between the injection of test datagrams and the appearance of encrypted messages in the reverse direction in the tunnel. This will be straightforward if the reverse direction is relatively quiet. If not, then false positives may occur. Then, to increase the accuracy of this approach, multiple trials can be used for each test datagram. We assume henceforth (without further mention) that this kind of technique is used in place of length-based detection whenever TFC measures are in force.

The attacker will use modifications of the ciphertexts C^j in his attack. In what follows, we let $C_0^j, C_1^j, \ldots, C_5^j$ denote the blocks of C^j (recall that C_0^j will be an IV).

4.2 Extracting the Rightmost Two Bytes

The attacker selects an arbitrary target block C_i, $i \geq 1$, from the target ciphertext. Consider the effect of injecting into the network an outer datagram containing as payload the ciphertext:

$$C' = C_0^6, C_1^6, C_2^6, C_3^6, R^6, C_i$$

where R^6 is a random block. The positioning of R^6 does not disturb the inner IP header in the corresponding plaintext P'. Thus the first part of P' will still correspond to an IP datagram carrying some upper layer protocol data destined for H_B. However, the payload data, padding, PL and NH bytes will be upset. Based on our discussions of ESP decryption and depadding and NH processing in Sections 2.3 and 3.1, we know that G_B will discard the inner datagram unless the padding is valid and the NH byte equals 4. With R^6 being random, by far the most likely valid padding pattern is the one of length zero, with the PL field containing 0. When this holds and the NH byte equals 4, decryption and depadding of C' produces a perfectly good IP datagram, albeit one whose payload data has been changed from C^6. The length and checksum fields in the IP header will still be correct and the TTL field will contain a value of 1. So, in this situation, the inner IP datagram will be passed to the IP implementation at G_B, and we will get an ICMP error message in the reverse direction on the tunnel. This message will be encrypted in a 9-block ciphertext, and the attacker can detect it based on its length.

It is now obvious that the attacker should systematically vary R^6 in bytes 6 and 7, inject the modified ciphertexts, and look for the tell-tale 9-block ciphertext in the reverse direction. When this is detected, the attacker knows that the last two bytes of $R^6 \oplus d_K(C_i)$ are equal to $0, 4$ with high probability. (A simple additional test, manipulating byte R_5^6, can be used to eliminate the unlikely case where the padding is valid but the PL byte does not equal 0.) From this, it is trivial to extract bytes $P_{i,6}, P_{i,7}$. The maximum number of trial injections needed is 2^{16}, and the average number 2^{15}.

We see that, using chosen plaintexts, we have established an "ESP trailer" oracle that can be used to extract bytes 6 and 7 of P_i simultaneously. The remaining bytes of P_i can be extracted using a fairly standard padding oracle attack, as described next.

4.3 Extracting the Remaining Bytes

The attacker can proceed to extract the remaining bytes much more efficiently, working in sequence from right ($P_{i,5}$) to left ($P_{i,0}$). The attacker now works with a ciphertext of the form:

$$C' = C_0^5, C_1^5, C_2^5, C_3^5, R^5, C_i$$

where R^5 is initially set to the value of R^6 that produced an ICMP message when extracting the rightmost two bytes, except that we put $R^5_6 = R^6_6 \oplus 1$. This ensures that P', the decrypted-and-depadded version of C', ends with bytes $1, 4$. Now the attacker systematically varies R^5 in byte 5 and injects the modified ciphertexts. Only when P' ends $1, 1, 4$ will we again see a 9-block ciphertext in the reverse direction. When this is detected, the attacker knows that byte 5 of $R^5 \oplus d_K(C_i)$ is equal to 1. From this, he can deduce the value of $P_{i,5}$.

Continuing in this way, it's now easy to see how the attacker can extract all the bytes of P_i, consuming an average of 128 and a maximum of 256 trials per byte, by using modified versions of ciphertexts C^j and extending the amount of padding one byte at a time.

4.4 Notes on the Attack

The complexity of the whole attack is at most $2^{16} + 6 \cdot 2^8$ trials per ciphertext block, and on average half this. The main term here arises from the need to extract the rightmost two bytes simultaneously. A term like this appears in all of the attacks in this paper. We have not been able to find a way to reduce it except in attacks against specific IPsec implementations where weaker padding checks are carried out (see Section 9 for details).

The attack works even if the IP implementation at G_B performs strict length checking: at each stage, the inner IP datagram produced after successful decryption and depadding has exactly the right number of bytes as indicated in its Total Length field. If relaxed checking is in operation, then we can reduce the number of chosen plaintexts needed from 7 to 1 by working just with the single plaintext P^0. This datagram has a Total Length field that indicates 12 payload bytes after the IP header, and ends with 7 bytes of padding and the NH byte. Now modifications to the last two blocks of C^0 made in the padding oracle attack produce either incorrectly padded plaintexts or plaintexts that are correctly padded but in which there are more bytes in the inner datagram than are indicated in the Total Length field. These plaintexts pass relaxed length checks.

The curious reader might wonder why we have not simply used ICMP echo request messages as our chosen plaintexts, with the idea that correct padding and NH byte would lead to the generation of an ICMP echo reply message in the reverse direction on the tunnel. This is a seductive idea, but it does not work in general. One problem is that the ICMP echo request messages include a checksum that covers the entire body of the ICMP message, and this body is upset by the introduction of random blocks during our attack. Thus the checksum would almost always be incorrect and the ICMP messages would simply be dropped before the reply message was generated. This checksum problem can be circumvented if relaxed length checking is assumed – we could then exploit ICMP echo request messages, and indeed it is not hard to see that a single chosen plaintext will do. However, we would need to make the additional assumption that a single SA is being used for both ICMP and non-ICMP traffic in this case. For these reasons, we have preferred to present a tutorial attack based on the TTL field.

This attack introduces the key ideas that we will use in the sequel. The main challenge now is to move from a chosen-plaintext attack to a ciphertext-only one.

5 An Attack Based on Options Processing

In this section, we present a ciphertext-only attack against tunnel-mode ESP that is based on options processing. We focus on the case of 64-bit blocks, sketching the 128-bit case at the end of this section. We assume that relaxed length checking is carried out by IP, sketching the modifications necessary to handle strict checks later. Otherwise, we make the same operating assumptions as in Section 4.

Our main idea is to take an existing ciphertext and transform it into a ciphertext carrying an inner datagram which always produces an ICMP error message because of a failure of options processing. The transformed ciphertext will then be used in a padding oracle attack similar to that above. We suppose the attacker has collected a number of target ciphertexts of the form $C = C_0, C_1, \ldots, C_q$ from the payload of an outer IP datagram that was directed towards G_B. This could represent traffic flowing from H_A to H_B, for example.

5.1 Preparation

Let C' denote one of the target ciphertexts from the tunnel. The preparation phase will be conducted just once on C', after which any non-IV block from any other target ciphertext can be decrypted efficiently. It will be convenient in practice to select a C' that is as short as possible. Let the blocks of C' be $C'_0, C'_1, \ldots, C'_r$, where we know that $r \geq 3$ (because C' must carry at least the header of an inner datagram, occupying 3 blocks).

The preparation stage can be codified as follows; an explanation follows.

1. Modify block C'_0 by flipping bits 6 and 7 to obtain a new block C''_0.
2. Set a counter i to 0.
3. **Repeat**:
   a. Modify block C''_0 so that bytes 4 and 5 contain the binary representation of i. Let $C^{\dagger}_0$ denote the modified block.
   b. Prepare an outer datagram directed to G_B with payload identical to C', except that C'_0 is replaced with $C^{\dagger}_0$. Inject this modified datagram into the network.
   c. Increment i.

   Until an ICMP-indicating ciphertext is seen on the tunnel from G_B to G_A.

We begin by flipping bits 6 and 7 of C'_0, the IV (here we number bits in a block from 0, starting at the left of a block). This has the effect of changing the IHL field in the inner header from 5 to 6, thus indicating that one 32-bit word of options are present in the inner header. The checksum calculation used by IP includes all bytes of the header, including any options bytes, so the checksum will now almost certainly be completely wrong. Thus we systematically modify bytes 4 and 5 of C'_0 in the loop, setting them to all possible 2^{16} values. This has the effect of modifying the Identification field in the inner datagram, as discussed in Section 2.8. We inject these modified ciphertexts into the network as the payloads of outer datagrams directed to G_B. After decryption and depadding, the resulting inner datagrams will be passed on to the IP implementation at G_B for further processing. Because of the modifications to the Identification field, exactly one of these datagrams will have a correct Header Checksum. It will also pass any length checks (relaxed or even strict). Options processing will then take place, with the first 4 bytes following the original inner header now being interpreted as options bytes. These will almost certainly be incorrectly formatted, leading to the generation of an ICMP type 12 "parameter problem" message. This message will be relayed through the reverse IPsec tunnel to host H_A, and will have a characteristic length of 9 blocks (assuming the ICMP RFC [19] has been followed and the inner datagram has at least 64 bits of payload).

We recall from [17] that, upon receipt of a datagram with random bytes in the Options field, an implementation of IP will produce an ICMP type 12 message with high probability. For example, this probability is 0.985 in Linux. Thus we can be almost certain that the attack above will succeed within 2^{16} trials, and on average using 2^{15} trials. In the unlikely event of the options bytes actually being correctly formatted, a different ciphertext can be tried in a fresh run of this stage.

At the end of this preparation stage, the attacker has in his possession a ciphertext C^* (say) with $r+1$ blocks that he knows will always lead to the generation of a detectable ICMP message when it is decrypted, depadded and passed to IP.

5.2 Extracting the Rightmost Two Bytes of a Block

The attacker is now ready to use the $r+1$ block ciphertext C^*, with blocks denoted by $C^*_0, C^*_0, \ldots, C^*_r$, to attack an arbitrary target ciphertext block C_i. He proceeds in a similar manner as in Section 4.2: he injects into the network an outer datagram containing as payload the $r+3$ block ciphertext:

$$C^{\#} = C^*_0, C^*_1, C^*_2, \ldots, C^*_r, R^6, C_i$$

where R^6 is a random block and C_i the target block. The plaintext blocks corresponding to these new blocks are interpreted as the last bytes of payload data, the padding bytes and the PL and

NH bytes. We know that the gateway G_B will discard the plaintext unless the padding is valid and the NH byte equals 4. By far the most likely valid padding pattern is the one of length zero, with PL byte 0. When the PL byte does equal 0 and the NH byte 4, decryption and depadding of $C^{\#}$ produces an inner datagram that is passed on to the IP implementation at G_B. The introduction of R^6 and C^i does not disturb the inner header. Thus we will have an inner datagram that has a correct Header Checksum and that has sufficiently many bytes to pass the relaxed length checks carried out by IP.[5] We see that the "old" padding, PL and NH bytes encrypted in blocks C^*_{r-1}, C^*_r are now regarded as redundant bytes by IP processing, being located beyond the last byte indicated by the Total Length field in the inner header. They are thus trimmed by IP. Because of the value of its IHL field, the inner datagram triggers options processing at G_B. And because of the content of the Options field, an ICMP message is generated in the reverse direction on the tunnel. The attacker can detect this based on its length.

As usual, the attacker systematically varies R^6 in bytes 6 and 7, injects the modified ciphertexts, and looks for the ICMP-carrying ciphertext in the reverse direction. When this is detected, the attacker knows that the last two bytes of $R^6 \oplus d_K(C_i)$ are equal to $0, 4$ with high probability. A simple additional test, manipulating byte R^6_5, can be used to eliminate the unlikely case where the padding is valid but the PL byte does not equal 0. From this, it is trivial to extract bytes 6 and 7 of plaintext block P_i. The maximum number of trial injections needed is 2^{16}, and the average number 2^{15}.

5.3 Extracting the Remaining Bytes

The attacker can proceed to extract the remaining bytes, roughly as in Section 4.3. The attacker now works with a ciphertext of the form:

$$C^{\#} = C^*_0, C^*_1, C^*_2, \ldots, C^*_r, R^5, C_i$$

where R^5 is equal to the value of R^6 that produced an ICMP message when extracting the rightmost two bytes, except that as before, we set $R^5_6 = R^6_6 \oplus 1$. This ensures that $P^{\#}$, the decrypted-and-depadded version of $C^{\#}$, ends with bytes $1, 4$. Now the attacker systematically varies R^5 in byte 5 and injects the modified ciphertexts. Only when P' ends $1, 1, 4$ will we again see an ICMP-indicating ciphertext in the reverse direction. When this is detected, the attacker knows that byte 5 of $R^5 \oplus d_K(C_i)$ is equal to 1. From this, he can deduce byte 5 of P_i.

Continuing in this way, it's now easy to see how the attacker can extract all the bytes of P_i, consuming an average of 128 and a maximum of 256 trials per byte.

5.4 Notes on the Attack

The complexity of the preparation phase is, on average, a little more than 2^{15} trials. The subsequent cost per block is of roughly the same complexity on average. A variant of the attack can also be applied in the case where the block cipher has 128-bit blocks The overall complexity is roughly the same as in the 64-bit case.

If strict length checking is being used by the IP implementation, then a much more complicated preparation stage is needed: we aim instead to produce from C^* a set of ICMP-generating ciphertexts $\{C^{*0}, C^{*1}, \ldots, C^{*6}\}$ in which C^{*j} encapsulates an inner datagram whose length equals j modulo 8, and which contains exactly the correct format for the padding bytes (with no redundant bytes after the payload being tolerated by the IP implementation). This set of ciphertexts can be built in two stages. First we find the extent of the padding in C^*, by modifying bytes from right to left starting in the second last block of C^*. If the padding is disturbed, then the padding format will be incorrect and IPsec will drop the data. On the other hand, if the modification is made to payload bytes located to the left of the padding, then the padding will be undisturbed and

[5] It is easy to show that there is no possibility that the introduction of R^6 and C_i produces a long enough padding pattern so that insufficient bytes remain after depadding to pass even the relaxed length checks subsequently carried out by IP.

an ICMP message will be generated. Then we simultaneously adjust the Total Length field, the Header Checksum field (both by manipulating the IV of C^*) and the padding bytes (by modifying the last-but-one block in C^*). The most efficient way to proceed involves an iterative approach in which the Total Length field is reduced by one and the padding pattern extended by one byte in each iteration. The details and the associated complexity analysis are omitted. If strict length checks are applied, we also need to be a little more careful in extracting plaintext bytes from the target ciphertext block. We work initially with variants of ciphertext C^{*6} when extracting bytes 6 and 7, and then with ciphertexts C^{*5} down to C^{*0} when attacking the other bytes. These ciphertexts contain inner datagrams whose Total Length fields have just the right values for extracting the targeted bytes.

6 An Attack Based on the Protocol Field

Similar ideas to those in the last section can be used to develop an attack which works in the 128-bit case by manipulating the Protocol field in the inner datagram.

The main idea is that an ICMP message will be generated if a host H_B receives a datagram whose Protocol field indicates an upper layer protocol that is not supported at H_B. Since protocol numbers from 138 to 252 are currently unassigned [13], it follows that inverting the two most significant bits of the protocol field will always result in the generation of an ICMP message.[6]

In a preparation phase, we capture a ciphertext, then modify its IV to flip bits in the protocol field and correct the checksum, with the aim of constructing an encrypted inner datagram that always results in an ICMP message. Since only two bit flips are involved, we can use tables of masks to correct the checksum; this results in a more efficient preparation phase which requires on average 7 trials for success (the analysis is the same as the protocol field attack in [17] which also involved two bit flips). The resulting datagram is then used in an attack which jointly extracts the last two bytes of each target ciphertext block, followed by the remaining bytes. The details of how this is done are identical to the attack based on options processing in Section 5, and the overall complexity is roughly the same (albeit with a much more efficient preparation stage).

One major difference between this Protocol field attack and the options processing attack in the previous section is that the former only works if a somewhat relaxed IPsec policy is deployed between G_A and G_B: we need the policy (or policies) to be protocol non-specific, i.e. to allow "ANY" protocol field to appear in inner datagrams. Otherwise, bit-flipping in the protocol field is likely to lead to a failure of policy checks during in-bound processing. Such liberal policies seem to be quite widely used in VPNs in practice, since VPNs are usually designed to protect all traffic flowing between gateways, irrespective of the traffic type.

We have not succeeded in developing a 64-bit version of this attack. The barrier is that the Protocol field is located in the second plaintext block and is not easily manipulated without upsetting further IP header fields. A substantially more complicated version of the 128-bit attack can be mounted when strict length checking is carried out; again, the details are omitted.

7 Attacking IPsec's Traffic Flow Confidentiality Mechanisms

Our techniques can be adapted to defeat the traffic flow confidentiality (TFC) methods specified in [12], in encryption-only mode. Recall from Section 2 that these comprise TFC padding and the use of an NH byte of 59 to indicate dummy packets. If these mechanisms are in use, it may be infeasible for an attacker to attempt a full decryption attack on every datagram he intercepts. The attacks in this section require much less effort than full decryption attacks. Using them, an attacker can quickly isolate the datagrams that contain real rather than dummy data, and then proceed with a full decryption attack against these. The attacks also demonstrate that the TFC mechanisms specified in [12] require the use of integrity protection in order to be effective.

[6] Provided that the original value of the protocol field is 60 or less. This includes both the TCP and UDP protocols which are of most interest.

Note that, since we assume TFC measures are being used in the IPsec tunnel, the correlation techniques mentioned in Section 4 will be needed to detect all ICMP messages generated during the attacks in this section.

7.1 Distinguishing Dummy Datagrams

Given a ciphertext with blocks $C = C_0, C_1, \ldots, C_q$, we can easily test whether it represents a dummy packet or not. We prepare a ciphertext $C^* = C_0^*, C_1^*, \ldots, C_r^*$ that is known to generate ICMP messages, through one of the methods described in Sections 5 or 6. Now we simply concatenate C^* and C, and inject the resulting $(r+q)$-block ciphertext $C' = C_0^*, C_1^*, \ldots, C_r^*, C_0, C_1, \ldots, C_q$ into the network. Note that the last byte of the decrypted version of C' equals the NH byte that was encrypted in the target ciphertext C. Moreover, if C was correctly padded, then so is C'. If the last byte of the decrypted version of C' has value 59, the inner datagram will be identified by the security gateway as a dummy, and it will be discarded before any ICMP message can be generated. On the other hand, if this last byte has value 4, then the decrypted payload will be depadded and forwarded to the IP layer, where it will be trimmed and eventually generate the expected ICMP message. Thus it is possible to discriminate genuine datagrams from dummy ones using a small number of trials per datagram after the preparation phase. Note that this attack does require relaxed length checking.

7.2 Defeating TFC Padding

We can always uncover the true length of the inner datagram that is encapsulated in a payload $C = C_0, C_1, \ldots, C_q$ (thus defeating the TFC padding mechanism) simply by decrypting the ciphertext block C_1 which contains the Total Length field. On the other hand, it can be possible to produce a more efficient attack with the same objective. We illustrate this in the 128-bit case, using a variant of our protocol field attack. We assume IP adopts relaxed length checking.

In a preparation phase, we capture an arbitrary ciphertext C' from the tunnel and use it to prepare a sequence of valid ESP trailers containing different amounts of padding. First, we manipulate C' using the method described in Section 6 to create a ciphertext $C^* = C_0^*, C_1^*, \ldots, C_r^*$ that always produces an ICMP message. Then we play with the last two blocks of C^* using an obvious modification of the padding oracle attack to create a set of 15 ciphertexts of the form:

$$C_i^* = C_0^*, C_1^*, \ldots, C_{r-2}^*, R_i, C_r^*, \quad 0 \le i \le 14$$

such that C_i^* is correctly padded and contains a padding pattern of length i, followed by a PL byte of i and an NH byte of 4. From C_{14}^*, we go on to create a special ciphertext of the form

$$C_{15}^* = C_0^*, C_1^*, \ldots, C_{r-3}^*, Q_{15}, R_{15}, C_r^*$$

such that C_{15}^* is correctly padded and contains a padding pattern of length 15, followed by a PL byte of 15 and an NH byte of 4. Here the padding pattern extends over the last 2 blocks. The total cost of this preparation phase is at most roughly $16 \cdot 2^8 = 2^{12}$ trials (because we begin with a correctly padded ciphertext, there is no step involving 2^{16} trials to set up a PL and NH combination of $0, 4$. This figure also assumes we can detect any ICMP messages based on time correlations without multiple trials).

Next, we take our target ciphertext $C = C_0, C_1, \ldots, C_q$, and create from it a $(q+1)$-block ciphertext that contains the same (encrypted) Total Length field and that always produces an ICMP message (again, using the method of Section 6). The cost of this is on average about 7 trials. Then we replace the last two blocks of this ICMP-generating ciphertext with blocks R_0, C_r^* to create a $(q+1)$-block ciphertext $C^{\#}$ containing a zero-length padding pattern. Because of relaxed length checking, $C^{\#}$ will still generate an ICMP message.

Now we systematically chop blocks out of $C^{\#}$, reducing the number of blocks in the ciphertext one at a time, and inject the resulting ciphertexts into the network. We begin by removing block

$C^{\#}_{q-2}$, then block $C^{\#}_{q-3}$, and so on. In this way, the injected ciphertexts are correctly padded and have the same ICMP-generating inner header as $C^{\#}$. When an ICMP message is no longer generated, we know that relaxed length checking has failed, in which case the number of bytes in the payload must now be less than are indicated in the Total Length field.[7] This tells us the value of the Total Length field up to the nearest block. This part of the attack can of course be speeded-up using a binary search.

Suppose the last ciphertext to generate an ICMP message is $C^{\dagger} = C^{\dagger}_0, C^{\dagger}_1, \ldots, C^{\dagger}_s$, with $s+1$ blocks. Then we know that the value of the Total Length field in the inner datagram lies between $16(s-2)+15$ and $16(s-1)+14$. Now we can exactly determine the Total Length field. Starting with $i = 1$, we replace the last two blocks of $C^{\dagger}$ with R_i, C^*_r for increasing values of i, and inject the resulting ciphertexts into the network. When $i = 15$, we replace the last three blocks of $C^{\dagger}$ with Q_{15}, R_{15}, C^*_r. Using the same reasoning as above, we see that ICMP messages will continue to be generated until the padding pattern is sufficiently long that relaxed length checking fails after depadding. For example if no ICMP message is generated for $i = 1$, then we can deduce that the value of the Total Length field in the inner datagram is $16(s-1)+14$. If we get to $i = 15$ and an ICMP message is still generated, then we can deduce that the Total Length field has its minimum value $16(s-2)+15$. This part of the attack can also be made more efficient using a binary search.

The overall number of trials needed in this length finding attack is (using the binary search variant and ignoring the preparation step) logarithmic in the ciphertext length. The cost of the preparation phase can be spread over many target ciphertexts. Even if we only attack a single ciphertext, include the cost of the preparation step, and assume multiple trials are needed to detect each ICMP message, the attack is still more efficient than simply decrypting block C_1.

We have presented the above attack in the 128-bit case. An obvious variant using our options processing ideas works in the 64-bit case. However, the attack is much less efficient: now every target ciphertext needs about 2^{15} trials on average to create an ICMP-generating variant, and it is roughly as efficient to decrypt C_1 directly.

8 Attacking Transport Mode ESP

Our ideas can be adapted to attack transport mode configurations of IPsec using encryption-only ESP. We sketch how this can be done in this section. We assume the reader is familiar with UDP.

We assume now that ESP is used in transport mode between a pair of hosts. As usual, we assume that the attacker can intercept ESP-protected datagrams flowing between the hosts, and inject modified datagrams into the network. We also make our usual assumptions about the presence of appropriate SAs.

In a preparatory phase, the attacker creates a set of datagrams that are used in the main attack. We assume the attacker is able to capture a datagram whose payload encapsulates a UDP message. Such datagrams can in theory be distinguished on the network because they do not result in acknowledgements, unlike TCP segments. The attacker then manipulates the payload of this datagram so as to create an encapsulated UDP message which always results in the production of an ICMP response from the end host receiving it, and which has a UDP checksum field that is equal to zero. The significance of the latter condition is that a checksum field containing zero will lead to the result of the checksum calculation being ignored by the processing host, so that the attacker need not worry about ever fixing the checksum in subsequent stages of the attack. This is very helpful since the UDP checksum covers the entire message and is therefore disturbed when, for example, target blocks are spliced onto a payload. The manipulation can be done by systematically varying fields of the IV corresponding to the UDP checksum field and the UDP port number field; the ICMP message produced will then be of the "port unreachable" type. These fields are contained in the first 64 bits of the UDP message, so this can be done even in the case where a 64-bit block cipher is in use. A simple additional test can be carried out to check that the

[7] We assume here that the content of the Total Length field is sufficiently large that we do not end up chopping out blocks containing bytes of the inner header.

checksum field contains zero (rather than the actual correct value): if the value is zero, then any further bit flip to the UDP message will still produce an ICMP message.

The attacker then discovers the true length of the encrypted UDP message, by modifying padding bytes in the last block(s) of the encrypted payload from right to left – once the modification affects UDP data rather than padding, ICMP messages will be seen once again. Then the attacker further manipulates the UDP payload, simultaneously varying the length field in the UDP header (again by manipulating the IV) and the contents of the padding bytes and the PL byte in the last block(s), to create a set of ICMP-generating encrypted UDP messages whose byte lengths cover a block. This set of messages has similar properties to the set of chosen plaintexts used in the tutorial attack in Section 4. A set is needed rather than a single message because UDP (at least in the OpenSolaris implementation) uses strict length checking, dropping messages if the number of bytes present in the message does not exactly match the content of the length field. The preparation phase is now complete.

The main part of the attack now largely follows the same steps as the attack in Section 4, with a target block being spliced onto the end of one of the pre-prepared encrypted UDP messages at each stage. Again, the rightmost two bytes need to be extracted jointly, costing on average 2^{15} trials, and this requirement dominates the cost of the attack as a whole.

Note that the attack as sketched can be used to recover non-UDP traffic encrypted using transport mode, providing it has been protected using the same SA as the initial UDP message.

9 Implementing the Attacks

To implement our attacks, we need to find a suitable implementation of IPsec conforming to the RFCs. More specifically, we need an implementation that inspects the ESP padding and discards the inner datagram in the event that it is incorrectly formatted. Because conformance testing is easier with access to source code, we have focused on open-source implementations of IPsec. Surprisingly, it was rather difficult to find an open-source implementation that conforms to the IPsec specifications.

9.1 Linux

The native Linux implementation of IPsec does not inspect the padding; it merely inspects the Pad Length field and chops off the number of bytes indicated there. Indeed the ESP decryption code contains the following comment at the point where a padding check should be made:

```
/* ... check padding bits here. Silly. :-) */
```

It is evident from this comment that the developers of Linux either do not understand the security implications of omitting the padding check, or they do not perceive the attack of Bellovin [3] as posing a serious threat. Recall that Bellovin's attack only works if the padding is not inspected, and only then on special length ciphertexts. However, this length condition is quite likely to be met in some applications, for example, when IPsec is used to protect telnet sessions. Moreover, the attack requires just 2^8 chosen plaintexts if variable length padding is permissable.

The lack of a padding check in Linux means that our attacks will not work directly. However, we can still recover the two rightmost bytes of every plaintext block using variants of our attacks. We next sketch how this can be done.

First, in a preparation phase, we capture a ciphertext from the tunnel and manipulate the inner datagram's header and the ESP trailer fields to produce two special ciphertexts C_1^* and C_2^* with the following properties. Ciphertext C_1^* with $r+1$ blocks has at least 256 more bytes of payload than are indicated in its Total Length field, and always produces an ICMP message. Ciphertext C_2^* with $s+1$ blocks ends with PL and NH bytes 0,4, has a Total Length field that indicates that the last payload byte immediately precedes the PL byte, and also always produces an ICMP message. These ciphertexts can be created using our usual techniques (with some extra

tricks to handle manipulation of the Total Length field for C_2^*), at a cost of a small multiple of 2^{16} trials in the 64-bit case, and at much lower cost in the 128-bit case.

We then use C^{*1} and C^{*2} to obtain the last two bytes of the plaintext P_i corresponding to an arbitrary ciphertext block C_i in a two-stage attack.

In the first stage, we inject ciphertexts of the form

$$C_0^{*1}, C_1^{*1}, \ldots, C_{r-2}^{*1}, R, C_i,$$

replacing the last block of C^{*1} with the target ciphertext block C_i and using a random block R. We then use a counter to make modifications to the last byte of R until an ICMP message is observed. When this occurs, we know that the decrypted version of the injected ciphertext ends with 4, because otherwise either a policy violation would occur, or (in implementations like Linux where policy checking is not carried out) the decrypted ciphertext would be passed to an upper layer protocol implementation other than IP that would most likely just drop the message. From this knowledge and C_{i-1}, we can easily find the rightmost byte of P_i. Notice that here we have ensured that enough extra bytes are present in C^{*1} so that depadding does not corrupt the inner datagram header, irrespective of the uncontrolled content of the PL field.

In the second stage, we use C^{*2} to extract the second-last byte of P_i. We inject ciphertexts of the form

$$C_0^{*2}, C_1^{*2}, \ldots, C_{s-2}^{*2}, R, C_i,$$

now replacing the last block of C^{*2} with C_i and re-using the successful value of R from the first stage. We now use a counter to make modifications to the second-to-last byte of R until an ICMP message is observed. When this occurs, we know that the decrypted version of the injected ciphertext ends with $0, 4$, because any longer valid padding pattern would result in an inner datagram with too few payload bytes to pass IP's relaxed length checks. (Recall that we set the Total Length field of C^{*2} to point to the payload byte immediately to the left of the PL byte.) From this, we can deduce the second last byte of P_i.

The worst-case cost of extracting the rightmost two bytes from each plaintext block is only 2^9 trials with this attack (2^8 trials for each byte). This attack then provides an example where it is possible to break through the "2^{16} barrier" that was imposed on our earlier attacks by the need to jointly extract the rightmost two bytes.

A simplified version of this attack was carried out against the Linux implementation of IPsec using a 128-bit block cipher (AES) and a test setup similar to that in [17]. Our implemented attack in fact extracts the rightmost two bytes jointly, essentially just using the second stage of the above attack. Here an ICMP message is generated only if the last two bytes of plaintext are correctly formatted with values $0, 4$, and the cost of the attack is at worst 2^{16} trials. The attack successfully recovered the last two bytes of every block, within 2^{16} trials.

Of course, better attacks are possible against the Linux implementation of IPsec – see [17]. But this experimental result illustrates that real attacks can be built using ESP trailer oracles. In turn this shows that our understanding of the operation of IPsec and IP processing is correct, so providing evidence that, if proper padding checks were carried out, then there should be no barrier to the operation of the full attacks in Sections 5 and 6.

9.2 Other Open Source Implementations

The FreeBSD and NetBSD IPsec implementations of IPsec are both the result of the KAME project. All three use a crude padding check: either the PL byte is zero or else it must be equal to the last padding byte. OpenBSD uses the same check, as does MacOS X (which makes use of the KAME IPsec implementation). This choice of padding check is somewhat justified as the ESP RFCs do not specify explicitly how the padding should be checked. The nature of this padding check means that all these implementations are vulnerable to the chosen-plaintext attack of Bellovin. They are also vulnerable to obvious variants of the ciphertext-only attacks in this paper: the attacker first arranges for the pattern 0,4 to appear at the end of the last plaintext

block, and then the pattern 1,1,4. These variants can extract the rightmost 3 bytes of any block using slightly more than 2^{16} trials.

Openswan and strongSwan, two independent continuations of the FreeS/WAN project, provide alternative IPsec implementations for Linux. Note that Openswan and strongSwan do not allow encryption-only configurations to be selected; nevertheless this policy is enforced by the key management daemon `pluto`, and it should therefore be possible to configure an encryption-only SA manually. Both do carry out a full padding check, but fail to drop the inner datagram in the event that the padding is incorrect. This is not inconsistent with the ESP RFCs, as these do not explicitly mandate datagram drops, but does immediately render them vulnerable to Bellovin's attack.

9.3 OpenSolaris

The OpenSolaris implementation of IPsec can be configured to perform a full padding check that drops datagrams, by adjusting a parameter `ipsecesp_padding_check`. If this parameter has value 0, the padding is not inspected at all, while the default value 1 indicates the KAME padding check should be carried out, and a value of 2 mandates a full padding check. To its credit, the OpenSolaris implementation does issue a warning to a user selecting an encryption-only configuration.

Our initial examination of the build 51 source code, reproduced in Appendix A, showed that the full padding check implementation was in fact incorrect: it produced an error when presented with length zero padding (where the PL and NH bytes are 0,4), unless the PL byte is itself preceded either with a zero byte or with another valid padding sequence of any length. Since these additional requirements refer to bytes from the payload of the inner datagram, and since the OpenSolaris code does allow zero-length padding to be selected during encryption, we had evidently discovered a bug in the OpenSolaris implementation. This bug resulted in legitimate datagrams being dropped during depadding, thus preventing correct communication between IPsec peers. Unfortunately, this bug also prevented our attacks from working. The bug was subsequently documented[8] and fixed in OpenSolaris build 55, after which we were able to successfully implement our attacks. We report on this next.

The experimental setup we used is essentially that of Figure 3, with an attack machine placed on the network between the two gateways. We focused on the protocol field attack of Section 6, with Gateway B running OpenSolaris build 55 and with Gateway A, Host A and Host B all running Linux kernel version 2.6.4. This setup meant that ICMP responses generated at Host B were contained in relatively long datagrams (since Linux follows [1] rather than [19] when implementing ICMP), but these were still detectable in the tunnel based on their lengths. We note that the production of ICMP messages is generally a rare event in our attacks, meaning that they are not particularly affected by ICMP rate-limiting. The VPN was configured to use AES with a 128-bit key. We ran the attack against 100 different ciphertext blocks (for a fixed key). The preparation phase required only a handful of trials. The average number of trials then needed to extract the rightmost two bytes of each block was 31343, while the average number of trials for each of the other 14 bytes was 128.6. The average number of trials to decrypt each complete block was 33144. These averages are in-line with our complexity analysis for this attack. The number of trials per second we could perform was limited only by the speed of our test network; in our setup, each block took only a few tens of seconds to decrypt.

9.4 Summary

It is evident that all of the open-source IPsec implementations we examined were deficient to some extent in their treatment of padding. Many are still vulnerable to Bellovin's attack [3], while others do not drop badly padded payloads. The single implementation that does full padding checks and drops failing payloads did not correctly implement the checks at first. Once this bug was fixed, our attack was successful. In summary, we have attacks that work only if implementors do properly

[8] `http://bugs.opensolaris.org/bugdatabase/view_bug.do?bug_id=6478675`

follow the RFCs, and attacks (from [3, 17] and variants of our padding oracle attacks) that work because implementors do *not* follow the RFCs.

10 Conclusions

We have presented attacks showing that encryption-only configurations of IPsec, as defined in and allowed by the IPsec standards, are highly insecure. These new attacks extend and complement the implementation-specific attacks in [3, 17]. Taken all together, the various attacks show that IPsec in encryption-only configurations is vulnerable, whether or not implementors follow the RFCs and carry out proper padding checks.

We consider it unfortunate that the IETF did not take the opportunity to outlaw encryption-only configurations once and for all in the new generation of IPsec RFCs (RFCs 4301-4309). Instead, we are reliant on implementors knowing what to do with warnings in the standards and on end-users to select their configurations appropriately. As was argued in [17, 18], the risk that end-users will actually select weak configurations is very real. A more conservative approach, which would seem to be appropriate in a security standard, might even have led to such a change after the original attacks of Bellovin were published [3]. However, the need for backwards compatibility seems to have over-ridden security concerns. It is then somewhat ironic that the IETF's selected solution to Bellovin's attacks – to recommend that padding checks should be carried out – is precisely what enables our new attacks.

In the light of our new attacks, some of the advice for implementors in [12] for when ESP might be usable in encryption-only configurations now appears to be rather ill-informed. For example, the idea that adequate security may be obtained "when higher-layer authentication is offered independently" is incorrect: our attacks work irrespective of any upper layer protection that might be applied, and some of our attacks work in certain configurations even if integrity protection is provided by AH. It appears that the IETF currently has no plans to further develop the IPsec standards; thus we seem to be stuck with such advice for some time to come.

The attacks in this paper also highlight some of the complexities inherent in properly specifying and then implementing secure communication protocols. For example, we have seen the tendency for implementors to ignore requirements to perform policy and/or padding checks stated in the IPsec RFCs, and how this reduces or enhances the applicability of attacks. (Nevertheless, our attack techniques are now sufficiently general that some variant will break almost any encryption-only configuration.) We've also seen how attacks like Bellovin's, well-known in the academic and IPsec standards communities, are either not taken seriously or not known about by implementors. Our analysis of various open-source implementations contradicts the often-made claim that an open-source approach results in better and more secure software than closed-source development. Indeed, by far the cleanest code we saw was that from OpenSolaris, which has only recently emerged from a closed-source environment. However, even that code failed to perform a full padding check correctly, and only had this check available as a non-default option. We note that similar observations concerning open-source security software were made in [16].

The research in this paper is located at the rather murky interface between cryptographic theory, security standards, and software development. It seems to us that this general area needs to be much better understood if we are to ensure that the useful outputs of theory are adequately carried forward into implementations. We certainly need to produce specifications that are much more precise about how to handle security-sensitive issues such as padding, policy checking and auditing, even if RFCs are traditionally aimed at ensuring inter-operability rather than security. For example, it is probably not enough to simply specify that the padding in ESP should be checked. Rather, it seems necessary to specify exactly how it should be checked, what the consequences of not checking it might be, and what actions should be taken if the checks fail. As we've seen, different implementors respond very differently to any vaguer prescription, making their software vulnerable to old attacks. Including explanations and references for why certain checks need to be performed would encourage better implementations and persuade implementors that the RFCs were not just being fussy. However, as we have shown, just getting padding checks right is not enough to prevent attacks on encryption-only ESP.

Many factors would reduce the reliability of our attacks in real networks, for example, ICMP blocking mechanisms in place at firewalls. However, it would seem unwise to rely on these weak protections to prevent our attacks. Ultimately, it seems better to be more defensive from the outset and outlaw configurations that are known to have security weaknesses.

Acknowledgements

We thank Dan MacDonald for his assistance with the OpenSolaris implementation of IPsec.

References

1. F. Baker, "Requirements for IPv4 Routers", RFC 1812, June 1995.
2. M. Bellare, T. Kohno and C. Namprempre, "Breaking and provably repairing the SSH authenticated encryption scheme: A case study of the Encode-then-Encrypt-and-MAC paradigm." *ACM Transactions on Information and System Security (TISSEC)*, Vol. 7, No. 2, May 2004, pp. 206–241.
3. S. Bellovin, "Problem Areas for the IP Security Protocols", in *Proceedings of the Sixth Usenix Unix Security Symposium*, pp. 1–16, San Jose, CA, July 1996.
4. N. Borisov, I. Goldberg and D. Wagner, "Intercepting Mobile Communications: The Insecurity of 802.11", in *Proc. MOBICOM 2001*, ACM Press, 2001, pp. 180-189.
5. B.Canvel, A.P. Hiltgen, S. Vaudenay and M. Vuagnoux, "Password Interception in a SSL/TLS Channel," in *D. Boneh (ed.), Advances in Cryptology – CRYPTO 2003*, LNCS Vol. 2729, Springer-Verlag, 2003, pp. 583–599
6. Cisco "Using Management Center for VPN Routers 1.3 — Working with Building Blocks", available from `http://www.cisco.com/en/US/products/sw/cscowork/ps3994/products_user_guide_chapter09186a00801f596a.html`. Last accessed 29th March 2007.
7. N. Doraswamy and D. Harkins. *IPsec: the new security standard for the Internet, Intranets and Virtual Private Networks (second edition)*, Prentice Hall PTR, 2003.
8. S. Frankel, K. Kent, R. Lewkowski, A.D. Orebaugh, R.W. Ritchey and S.R. Sharma, "Guide to IPsec VPNs", NIST Special Publication 800-77, Dec. 2005.
9. S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol", RFC 2401, Nov. 1998.
10. S. Kent and R. Atkinson, "IP Encapsulating Security Payload (ESP)", RFC 2406, Nov. 1998.
11. S. Kent and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301 (obsoletes RFC 2401), Dec. 2005.
12. S. Kent, "IP Encapsulating Security Payload (ESP)", RFC 4303 (obsoletes RFC 2406), Dec. 2005.
13. Internet Assigned Numbers Authority (IANA), "Assigned Internet Protocol Number." Available from `http://www.iana.org/assignments/protocol-numbers`.
14. Internet Protocol, RFC 791, Sept. 1981.
15. C.B. McCubbin, A.A. Selcuk and D. Sidhu, "Initialization vector attacks on the IPsec protocol suite." In *9th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2000)*, IEEE Computer Society, 2000, pp. 171–175.
16. P.Q. Nguyen, "Can we trust cryptographic software? Cryptographic flaws in GNU Privacy Guard v1.2.3", in *C. Cachin (ed.), Advances in Cryptology – EUROCRYPT 2004*, LNCS Vol. 3027, Springer-Verlag 2004, pp. 555–570.
17. K.G. Paterson and A.K.L. Yau, "Cryptography in theory and practice: The case of encryption in IPsec." In *S. Vaudenay (ed.), Advances in Cryptology – EUROCRYPT2006*, LNCS Vol. 4004, Springer-Verlag, 2006, pp. 12-29. Full version available at `http://eprint.iacr.org/2005/416`.
18. K.G. Paterson and A.K.L. Yau, "Lost in translation: theory and practice in cryptography." *IEEE Security and Privacy*, Vol. 4, No. 3, May/June 2006, pp. 69-72.
19. J. Postel, "Internet Control Message Protocol", RFC 792, Sept. 1981.
20. S. Vaudenay, "Security flaws induced by CBC padding – applications to SSL, IPSEC, WTLS...", in *L.R. Knudsen (ed.), Advances in Cryptology – EUROCRYPT 2002*, LNCS Vol. 2332, Springer-Verlag 2002, pp. 534–545.
21. T. Yu, S. Hartman and K. Raeburn, "The perils of unauthenticated encryption: Kerberos version 4", in *Proc. NDSS 2004*, The Internet Society, 2004.

Appendix A: OpenSolaris full padding check source code

We reproduce below the build 51 OpenSolaris 'C' source code which is responsible for carrying out the full padding check. At the start of this code, the variable `last` points to the byte before the Pad Length byte (ostensibly, the last byte of padding, but actually a byte of payload when the Pad Length byte contains zero). It can be seen that this code does not properly handle the case where the content of the Pad Length field is zero: the main loop between lines 914 and 926 only exits without error if the byte before the Pad Length field is also zero, or if the bytes prior to the Pad Length field form a valid padding pattern of some non-zero length. This code was replaced in build 55.

```
902     if (ipsecesp_padding_check > 1) {
903         uint8_t *last = (uint8_t *)(scratch->b_wptr - 3);
904         uint8_t lastval = *last;
905 
906         /*
907          * this assert may have to become an if
908          * and a pullup if we start accepting
909          * multi-dblk mblks. Any packet here will
910          * have been pulled up in esp_inbound.
911          */
912         ASSERT(MBLKL(scratch) >= lastval + 3);
913 
914         while (lastval != 0) {
915             if (lastval != *last) {
916                 ipsec_rl_strlog(info.mi_idnum, 0, 0,
917                     SL_ERROR | SL_WARN,
918                     "Possibly corrupt ESP packet.");
919                 esp1dbg(("padding not in correct"
920                     " format:\n"));
921                 ESP_BUMP_STAT(bad_padding);
922                 *counter = &ipdrops_esp_bad_padding;
923                 return (B_FALSE);
924             }
925             lastval--; last--;
926         }
927     }
```