802.11 Security Series

Part II: The Temporal Key Integrity Protocol (TKIP)

Jesse Walker
Network Security Architect, Platform Networking Group
Intel Corporation

intel.

I'

Agenda

This is the second of a series of three articles surveying security for IEEE 802.11 wireless LANs (WLANs) [1]. Part II examines the *Temporal Key Integrity Protocol*, or *TKIP*, a suite of algorithms likely to be put into place to patch WEP (Wired Equivalency Protocol) on legacy 802.11 equipment.

Review of Part I

To set the context, recall that Part I reviewed 802.11 and the problems surrounding WEP, the original WLAN security protocol. The major WEP flaws may be summarized into four categories:

- WEP provides no forgery protection. Even without knowing the encryption key, an adversary can change 802.11 packets in arbitrary, undetectable ways, deliver data to unauthorized parties, and masquerade as an authorized user. Even worse, an adversary can also learn more about the encryption key with forgery attacks than with strictly passive attacks.
- WEP offers no protection against replays. An adversary can create forgeries without changing any data in an existing packet, simply by recording WEP packets and then retransmitting later. Replay, a special type of forgery attack, can be used to derive information about the encryption key and the data it protects.
- WEP misuses the RC4 encryption algorithm in a way that exposes the protocol to weak key attacks, and public domain hacker tools like AirSnort exploit this weakness. An attacker can utilize the WEP IV to identify RC4 weak keys, and then use known plaintext from each packet to recover the encryption key.
- By reusing initialization vectors, WEP enables an attacker to decrypt the encrypted data without ever learning the encryption key or even resorting to high-tech techniques. While often dismissed as too slow, a patient attacker can compromise the encryption of an entire network after only a few hours of data collection.

It is worth asking whether a patch must address all of these categories of problems, or whether an adequate solution can resolve only some of them. Analysis always leads to the same conclusion: all the areas must be addressed to obtain privacy, even against casual hacking. This seems harsh and counter-intuitive, since incremental improvement in most areas offers real benefit. However, WEP does not address all these functions, and to achieve privacy a packet protocol must defend against all these types of compromise. A partial patch addressing just some of the problems is like adding a span or two to a bridge missing all of its spans. Such a bridge still fails to satisfy its most basic functional requirement of providing continuous passage to the farthest shore, and will continue to do so until all the spans are put in place.

Problem constraints

An army seeks battlefields offering the greatest possible advantage over its enemy, and avoids battlefields threatening peril. Task Group i, or TGi, established by IEEE 802.11 to resolve the security problems with WEP, is no different. WEP is so seriously flawed that TGi's most promising avenue is to create a new security protocol from scratch. While good security protocols are difficult to design, this is the surest road to success, because a new design would not be constrained by the WEP design. Indeed, TGi is developing an all-new security protocol for 802.11, and we will discuss this in next month's installment.

Millions of WEP-based devices, however, have already shipped and deployed, and the industry has an obligation to correct the security defects it inserted into this installed base if at all possible. Like most modern communication equipment, 802.11 devices are comprised of hardware and software. WLAN hardware has been designed as a commodity, so it is not cost effective to add or swap out particular hardware chips in a WLAN device. Instead, it is cheaper to replace the hardware as an entire unit. This implies that ***WEP patches operating on already-deployed 802.11 hardware will, of necessity, rely entirely on software upgrade. This is the first design constraint,*** and it poses a particularly sticky dilemma.

While stations may pose their own performance problems, access points present the fundamental computational bottleneck in the system, as they have little spare CPU capacity. Recall that in an infrastructure deployment, all stations link with the access point instead of communicating directly with each other, and the access point handles every message exchanged across the wireless network. In order to be competitive, commodity access points are typically implemented with the cheapest microprocessor possible – perhaps an i486™ microprocessor, ARM7, or PowerPC™ running at 40 or even 25 MHz. The load generated by normal WLAN traffic often consumes 90% or more of the microprocessor bandwidth. Thus, very few spare cycles are available for new functions, with as few as 2 million instructions per second unused on lower-end access points. ***This paucity of spare cycles is the second major design constraint.***

This is an impassible barrier for a traditional security design targeted at software. The cryptographic functions such a design would employ are some of the most CPU-hungry algorithms ever devised. Studying the design of a typical cryptographic algorithm can help clarify the need for CPU cycles. A cryptographic primitive, more often than not, consists of a simple function providing weak security, iterated many times, with each iteration magnifying the security effects of the previous one. The cryptographic operation applies this iterated operation to each byte in the data stream, so the cost of the algorithm is the per-byte cost times the number of bytes in the data stream. As an example, a popular "C" language public domain implementation of the 3DES encryption algorithm has an amortized cost of about 180 instructions per byte. With actual 802.11b user data throughput—around 7 million Mbps = 875,000 bytes/sec—this implementation requires about $180 \times 875000 = 157.5$ million instructions per second, roughly an order of magnitude in excess of an access point's total CPU budget for *all* functions, not just for new security features! While many other standard cryptographic primitives are much less expensive, all consume far more CPU cycles than are available on shipping access points.

On first analysis, then, fixing WEP with any cryptographically sound approach seems to be impossible without instantly obsolescing all existing hardware—which is what the proposed long-term solution described next month would do. It is impossible to use standard cryptographic functions to rescue WEP, at least on already-deployed hardware, because none of the equipment has sufficient spare CPU

capacity to accommodate the needed operations. The only alternatives within the present hardware base are to do nothing or to design new, custom cryptographic algorithms, and hope for the best.

It is worth noting that access points, since they support WEP, already implement RC4 encryption. How is this possible on such CPU-constrained devices? The answer is that nearly all shipping access points that support WEP have custom hardware to off-load the encryption function from the CPU. Most of this hardware is tuned to construct per-packet keys according to the WEP algorithm. The per-packet key is a base key concatenated to an initialization vector, or IV, which appears as plaintext in each packet. On transmit, the hardware expects the packet as its input, along with the base key and IV. It then constructs the per-packet key, encrypts the packet data payload, inserts the IV into the packet, and passes the result to the radio transmitter. On receive, the hardware removes the IV from the packet, locates the base key, constructs the per-packet key, and decrypts the packet payload as it arrives from the radio receiver. The duty cycle of most receivers is such that there is time to execute at best only two- or three hundred instructions between packet arrival and when the decryption engine consumes the packet, IV, and base key. ***The hardwired encryption function with its hard-wired per-packet encryption key construction, as well as its location in the data path, represents a third major design constraint.*** The design affords few opportunities for software intervention into an outgoing packet after encryption or into an arriving packet prior to decryption.

TKIP and its components

TKIP is TGi's response to the need to do something—anything—to improve security for already-deployed 802.11 equipment. TGi has proposed TKIP as a mandatory-to-implement security enhancement for 802.11, and patches implementing it will likely be available for most equipment in late 2002.

TKIP is a suite of algorithms wrapping WEP, to achieve the best security that can be obtained given the problem design constraints. The TKIP algorithms are designed explicitly for implementation on legacy hardware, hopefully without unduly disrupting performance. TKIP adds four new algorithms to WEP:

- A cryptographic *message integrity code*, or *MIC*, called Michael, to defeat forgeries;
- A new *IV sequencing discipline*, to remove replay attacks from the attacker's arsenal;
- A per-packet *key mixing function*, to de-correlate the public IVs from weak keys; and
- A *rekeying* mechanism, to provide fresh encryption and integrity keys, undoing the threat of attacks stemming from key reuse.

The remainder of this section analyzes each of the TKIP components, and the next section indicates how they are intended to work together to rescue WEP.

TKIP is an acronym for "Temporal Key Integrity Protocol." The name is something of a misnomer. The TKIP rekeying mechanism updates what are called temporal keys, which are consumed by the WEP encryption engine and by the Michael integrity function.

Defeating forgeries: Michael

A MIC is cryptographic device to detect forgeries. The literature calls these *message authentication code*s, or *MAC*s. Since IEEE 802 had already appropriated the acronym MAC to mean "media access control," TGi uses the MIC instead. Classical MICs include the CBC-MAC, constructed from a block cipher and used widely in banking applications, and HMAC, used by Ipsec (Internet Protocol Security).

Every MIC has three components: a secret authentication key K (shared only between the sender and receiver), a tagging function, and a verification predicate. The tagging function takes the key K and a message M as its inputs, and generates a tag T, also called the message integrity code, as its output. A protocol protects a message M from forgery by having the sender compute the tag T and send it with the message M. To check for a forgery, the receiver inputs K, M and T into the verification predicate. The predicate evaluates to TRUE if the reputed tag T is what should have been produced by the tagging algorithm, and FALSE otherwise. If verification indicates FALSE, the message is a failed forgery. If the verification function returns TRUE, the message is presumed authentic. A MIC is considered secure if it is infeasible for an attacker to select the correct tag for some new, never-seen-before message M, without knowing the key K.

Michael [2] is the name of the TKIP message integrity code. It is an entirely new MIC designed by Niels Ferguson.

The Michael key is 64-bits, represented as two 32-bit little-Endian words (K_0,K_1).

The Michael tagging function first pads a message with the hex value 0x5a and enough zero pad to bring the total message length to a multiple of 32-bits, then partitions the result into a sequence of 32-bit words $M_1\ M_2 \ldots M_n$, and finally computes the tag from the key and the message words using a simple iterative structure:

$(L,R) \leftarrow (K_0,K_1)$
do i **from** 1 **to** n
  $L \leftarrow L \oplus M_i$
  $(L,R) \leftarrow b(L,R)$
return (L,R) as the tag

where $\oplus$ denotes exclusive or (XOR) and b is a simple function built up from rotates, little-Endian additions, and bit swaps.
The Michael verification predicate reruns the tagging function over the message and returns the result of a bit-wise compare of this locally computed tag and the tag received with the message.

Performance is one of the dominating concerns in Michael's design and the hardest issue facing TKIP. Michael's inner loop—the function *b* above—uses only XORs, shifts, byte-swaps, and additions. Michael costs about 3.5 cycles/byte on an ARM7, and about 5.5 cycles/byte on an i486 processor. This means that, at 802.11b rates, it will consume about 3.1 M cycles/sec on an ARM7-based access point, and 4.8 M cycles/sec on an i486 processor based access point. This by itself ought to consume every unused cycle of many first-generation 802.11 access points, so some performance degradation should be expected. However, no cheaper alternative exhibiting appropriate security characteristics is known.

Security is the other lead issue with Michael. The security level of a MIC is usually measured in bits. If the security level of a MIC is *s* bits, then, by definition, the time required for an attacker to construct a forgery is, on average, after about 2^{-s+1} packets.

It is easy to establish an absolute upper bound on the security afforded by any MIC, secure or not. If *n* bits represent its tag, then the tagging algorithm maps any message to one of 2^n possible tags. This means that an attacker could create a forgery by sending any message of its choice 2^n *times*, once with each of the 2^n possible tags. By this reasoning we know that $s \le n$ for any real MIC, with equality obtaining when guessing is the best possible attack. For a real MIC, attacks better than guessing are normally possible. Michael was designed with a target security level of 20 bits. The best attack known against Michael relies on differential cryptanalysis and uses 2^{29} messages, so for Michael, 29 bounds *s*, not the 64 the MIC size might lead one to expect. This makes Michael a much weaker cryptographic primitive than what one normally wants.

On an 802.11b WLAN, an attacker could theoretically sustain a rate of 2^{12} small packets per second. This means that an adversary could expect to create a forgery against Michael after about 2^7 seconds worth of messages (assuming a 20-bit security level), or somewhat over two minutes. The corresponding numbers for 802.11a are 2^{13} packets and 2^6 seconds, respectively. This level of protection is much too weak to afford much benefit by itself, so TKIP complements Michael with counter-measures. The design goal of the counter-measures is to throttle the utility of forgery attempts, limiting the knowledge the attacker gains about the MIC key. If a TKIP implementation detects two failed forgeries in a second, the design assumes it is under active attack. In this case, the station deletes its keys, disassociates, waits for a minute, and then reassociates. While this disrupts communications, it is necessary to thwart active attack. The countermeasures algorithm thus limits the expected number of undetected forgeries such an active adversary might generate to about one per year per station.

Defeating replays: IV sequence enforcement

One forgery a MIC cannot detect is a replayed packet. This occurs when an adversary records a valid packet in flight and later retransmits it. The standard way to address this problem is to associate a packet sequence number space with a MIC key, and to reinitialize the sequence space whenever the MIC key is replaced. This strategy requires that the transmitter refrain from sending data protected by the old MIC key once it exhausts the sequence number space. The choices available to the transmitter on exhaustion are (a) halt communications altogether, (b) try to rekey the MIC with a fresh key, or (c) send subsequent traffic without any cryptographic protections. Failing to adopt one of these strategies risks exposing the data already protected under the key.

TKIP closely follows this classical design. To defeat replays, TKIP reuses the WEP IV field as a packet sequence number. Both transmitter and receiver initialize the packet sequence space to zero whenever new TKIP keys are set, and the transmitter increments the sequence number with each packet it sends. TKIP requires the receiver to enforce proper IV sequencing of arriving packets. TKIP defines a packet as out-of-sequence if its IV is the same or smaller than a previous correctly received MPDU associated with the same encryption key. If an MPDU arrives out of order, then it is considered to be a replay, and the receiver discards it and increments a replay counter. [3] summarizes TKIP replay protection rules.

TKIP deviates somewhat from the usual design by associating the sequence number with the TKIP encryption key, not with its MIC key. This constraint derives from the choice of the WEP IV field as the sequence space. This was done in order to reuse the existing WEP hardware and hence packet formats. The 1999 IEEE 802.11 standard associates the IV field with packet fragments, or MPDUs, whereas legacy access points will of necessity calculate the TKIP MIC over the entire packet, or MSDU. We will explain why this works below, in the section called "Putting the pieces together."

TKIP replay detection has at least one serious limitation. It will not work with the quality of service (QoS) enhancements being proposed by IEEE 802.11 Task Group e. It will be necessary to alter at least the enforcement step at the receiver to accommodate QoS, and perhaps the IV selection algorithm as well. The extension to properly detect replays in the presence of QoS is not yet defined.

Defeating weak key attacks: key mixing

Unlike a MIC, replay detection, or rekeying, all of which are mandatory for any protocol claiming to provide privacy, TKIP's per-packet key construction is a feature uniquely necessary to correct WEP's misuse of RC4. Recall that WEP constructs a per-packet RC4 key by concatenating a base key and the packet IV. The new per-packet key construction, called the TKIP key mixing function, substitutes a *temporal key* for the WEP base key and constructs the WEP per-packet key in a novel fashion. Temporal keys are so named because they have a fixed lifetime and are replaced frequently.

Doug Whiting and Ron Rivest invented the TKIP key mixing function. It transforms a temporal key and packet sequence counter into a per-packet key and IV. [4] specifies the key mixing function in detail. The mixing function operates in two phases, with each phase compensating for a particular WEP design flaw. Phase 1 eliminates the same key from use by all links, while Phase 2 de-correlates the public IV from known the per-packet key.

Phase 1 combines the 802 MAC address of the local wireless interface and the temporal key by iteratively XORing each of their bytes to index into an S-box[1], to produce an *intermediate key*. Stirring the local MAC address into the temporal key in this way causes different stations and access points to generate different intermediate keys, even if they begin from the same temporal key—a situation common in ad hoc deployments. This construction forces the stream of generated per-packet encryption keys to differ at every station, satisfying the first design goal. The Phase 1 intermediate key must be computed only when the temporal key is updated, so most implementations cache its value as a performance optimization.

Phase 2 uses a tiny cipher to "encrypt" the packet sequence number under the intermediate key, producing a 128-bit per-packet key. In actuality, the first 3 bytes of Phase 2 output corresponds exactly to the WEP IV, and the last 13 to the WEP base key, as existing WEP hardware expects to concatenate a base key to an IV to form the per-packet key. This design accomplishes the second mixing function design goal, by making it difficult for an adversary to correlate IVs and per-packet keys.

The tiny cipher defined for Phase 2 has a *Feistel structure*, which means its inner loop implements a transformation of the form $(L, R) \rightarrow (R, L \oplus f(R))$. As with Michael, the tiny cipher's inner loop can be implemented using only simple operations: XORs, shifts, rotates, and table look-ups - all cheap operations for processors commonly in 802.11 devices. Phase 2 represents the packet sequence number as a 16-bit little-Endian counter. Phase 2 assigns the 8 most significant bits of the counter to the first and second bytes of the WEP IV, and the least significant counter bits to the third IV byte. It then masks off the most significant bit of the second IV byte to prevent the WEP per-packet key concatenation from producing one of the known RC4 weak keys.

Performance considerations dominated the mixing function design, with the duty cycle of the receive path placing a hard cap on the number of CPU cycles it can consume. The mixing function per-packet cost is about 150 cycles, situating it at the bleeding edge of what can be achieved within the context of existing access point hardware.

The security analysis of the key mixing function is somewhat less satisfying than that for Michael, as no quantitative bound such as 29-bits is known. However, review by the cryptographic community thus far suggests that it does indeed achieve its design goals, and that it is close to an optimal solution for the problem WEP poses.

Defeating key collision attacks: rekeying

The definition of the last TKIP element, rekeying, is not yet complete. Rekeying delivers the fresh keys consumed by the various TKIP algorithms. However, the general outline of this scheme can be described.

The TGi rekey architecture will depend upon a hierarchy of at least three key types: temporal keys, key encryption keys, and master keys. Occupying the lowest level of the hierarchy are the temporal keys consumed by the TKIP privacy and authentication algorithms proper. TKIP employs a pair of temporal key types: a 128-bit encryption key, and a second 64-bit key for data integrity. TKIP uses a separate pair of temporal keys in each direction of an association. Hence, each association has two pairs of keys, for a total of four temporal keys. TKIP identifies this set of keys by a two-bit identifier called a *WEP keyid*.

Previously we observed that WEP IVs can never be reused with the same key without voiding the RC4 privacy guarantees, and that the TKIP key mixing function can construct at most 2^{16} IVs. This implies that TKIP requires a key-update mechanism operating at least every 2^{16} packets. This is accomplished using the mechanism described below. To affect rollover from one set of keys to another, each association allocates two WEP keyids. When an association is first established, a first set of temporal keys is bound to one of these two WEP keyids. As new keys are created, the association ping-pongs between the two keyids, with the new set of temporal keys being bound to the least-recently bound keyid. After binding the new set of temporal keys, TKIP implementations still receive packets on the old key id and its temporal keys, but subsequently transmit only under the new keyid and its keys. The requirements for new temporal keys are that they are fresh—that is, it is unlikely that any have been used, even with a prior session with the same or another peer, even across reboots—and there is no algorithmic relationships among the keys in the set.
The instantiation of new temporal keys requires careful coordination. TGi accomplishes this using special rekey key messages. The rekey key message distributes keying material from which both the station and AP derive the next set of temporal keys. This exchange must itself be secure, or an attacker can compromise the temporal keys by compromising the rekey key message. The next level of the hierarchy, the key encryption keys, protects the temporal keys. There are two key encryption keys: one to encryption the distributed keying material, and a second to protect the rekey messages from forgery. The requirements for key encryption keys are similar to those for temporal keys: the station and access point must establish a fresh set on association or reassociation.

One architecture designed to accomplish these requirements relies on the 802.1X and uses the 802.1X authentication server to push a common set of key encryption keys to the station and the access point. Under the 802.1X architecture, the authentication server and access point already share a key that can protect one half of this distribution. All that is needed is some way to secure the other half of the push, one between the authentication server and the station. This leads to the highest level of the key hierarchy, the master key. This is a key shared between the station and the 802.1X authentication server. This is the key most directly tied to the authentication, and is used to secure the distribution of the key encryption keys. The master key is established as a side effect of authentication. A natural session structure can be formed based on this key, spanning from authentication until the key is revoked, expires, or the station loses contact with the infrastructure. The requirements for the master key are that it is fresh over each such session, and that the master key for any one session is unrelated to the master key for any other session. If either of these conditions is violated, an attacker can more easily compromise the master key and then trivially the key encryption and temporal keys, thus voiding any TKIP privacy claims.

---

[1] An S-box is an invertible non-linear substitution table.

Some open issues related to rekeying

This general outline of rekeying is necessary but not sufficient to provide TKIP security. Other problems must be resolved.
One major issue is how to support roaming by stations from one access point to another in large infrastructure deployments. A full reauthentication by the 802.1X authentication server can be too slow to support real-time applications like voice over IP traffic, so some faster mechanism may be required. This suggests some sort of key passing architecture. Key passing is especially prone to attack, so the design of this type of architecture is quite delicate. TGi is still investigating this problem.

A second issue is how to extend the rekey architecture into ad hoc networks. The current architecture assumes a session-oriented structure—initial contact association is used to synchronize the master key, and the association notion is also used to subsequently synchronize the key encryption keys and the security mechanisms to enforce. Finally, 802.1X assumes an authentication server. None of these mechanisms exist in an ad hoc network. They will either have to be added, the security architecture altered, or security will not be possible in an ad hoc network.
A final problem is how to distribute and use multicast/broadcast keys. These must be updated "simultaneously" at all stations, so the unicast key distribution message is poorly suited for this function. On the other hand, radio is inherently prone to signal deterioration, meaning that using broadcast/multicast itself to distribute temporal key updates is problematic.

Putting the pieces together

Now that we have seen all of the pieces of TKIP, we can see how they work together with WEP to provide security.
When a station makes an initial contact exchange with an infrastructure, it uses 802.1X to authenticate and derive a fresh master key. The master key remains in use until it expires or is revoked. The master key is tied directly to the authentication step, so authorized by it, and this key implicitly authorizes all the subsequent keys and all traffic they protect.

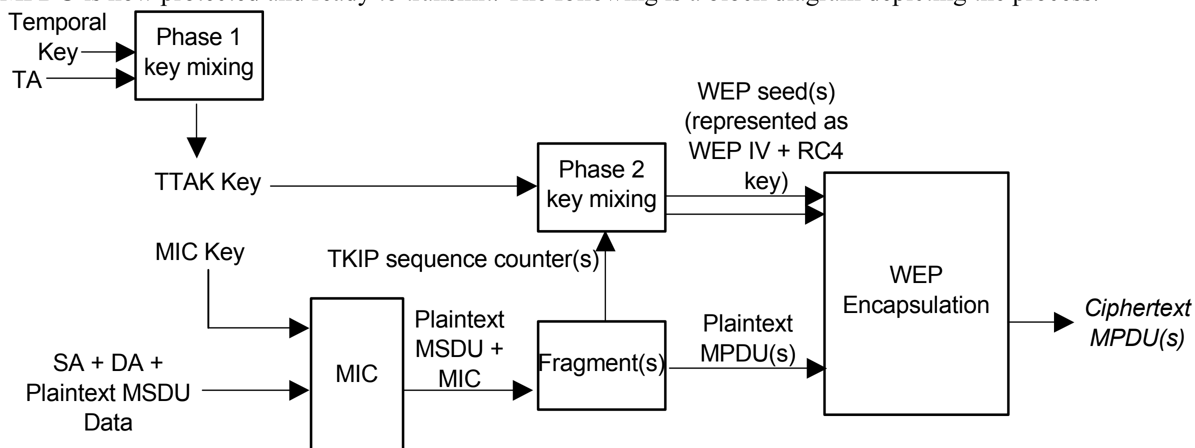
After establishing a master key, the authentication server distributes fresh keying material to the station and the access point to derive a first set of key encryption keys. The master key protects the exchange between the authentication server and the station. The protocol also provides a means for the station and AP to update themselves with fresh key encryption keys after every reassociation.

The access point uses the established key encryption keys to protect fresh temporal keys it sends to the station. The access point repeats the key refresh message slightly more frequently than once every 2^{16} data packets, so that the association will always have keys and associated packet sequence spaces meeting the algorithm assumptions.

When security is in use, neither the access point nor the station permits normal data traffic until the temporal keys are in place. Once this is accomplished, the access point and station can use the temporal keys to protect packets using TKIP, and both discard non-TKIP-protected traffic if any is received.

The TKIP encapsulation and decapsulation processes are simple to describe.

When the station wishes to transmit an MSDU, the TKIP implementation uses the temporal Michael key to compute the MIC of the source and destination MAC addresses, as well as the MSDU payload. TKIP appends the MIC to the data field, thus extending the packet's data payload by 8 bytes. Next the 802.11 implementation fragments the MSDU into MPDUs as needed by the ambient environment. Once done, it assigns each fragment a packet sequence number and employs the key mixing function to create a per-packet encryption key for each, represented as a WEP IV and a base key. At this point, the remaining steps are pure WEP, usually implemented in hardware. The system computes and appends the ICV to the data field ICV of each fragment. The encryption consumes the IV and base key, encrypts the data field, including the MIC and ICV, and encodes the IV and the key id of the set of temporal keys into the WEP IV field, completing the encapsulation process. The entire MPDU is now protected and ready to transmit. The following is a block diagram depicting the process.

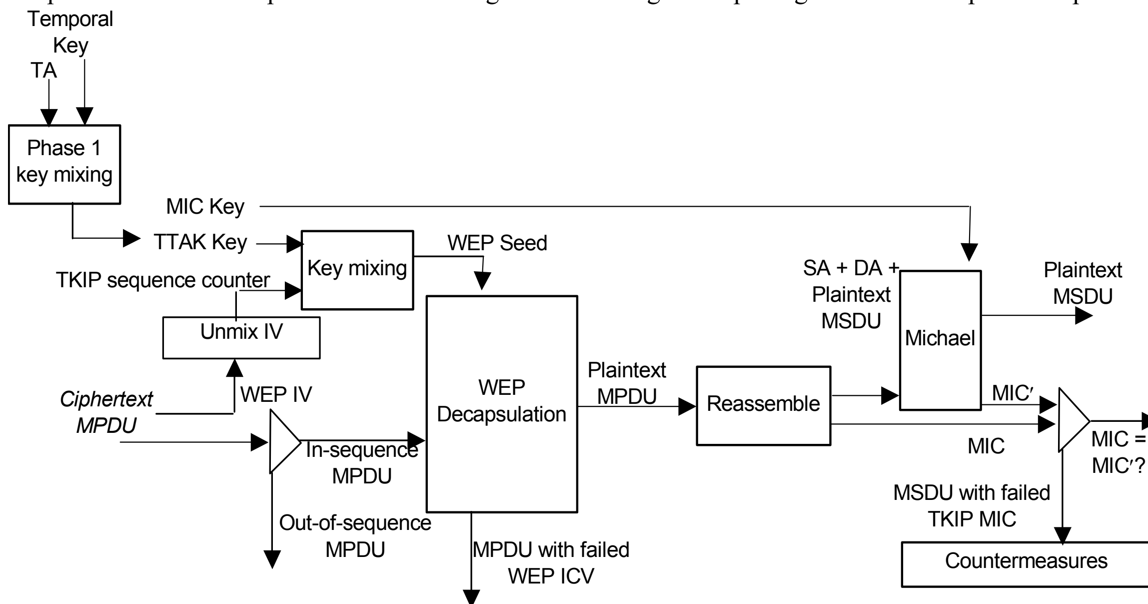


Note that since TKIP uses a single key pair to compute the MIC and to encrypt each of the fragments, it is necessary for the sender to be able to predict the number of fragments that each MSDU will generate. In particular, if the number of remaining sequence numbers cannot cover all

the fragments, then the sender must select a new set of temporal keys prior to encapsulation. Without this step the receiver can select the wrong key at the receiver, causing the MIC verification to fail.

The inter-relationship of the pieces of the TKIP encapsulation is believed to enhance its overall security. The TKIP design uses RC4 encryption to encrypt the MIC. It is conjectured that this decreases the amount of information about the MIC key available to an attacker. An attack that changes the IV also changes the per-packet temporal encryption key, making it likely that both the ICV and MIC will decrypt incorrectly. The MIC makes bit-flipping attacks that alter the encrypted data and ICV in a coordinated fashion more difficult. Since the MPDU is protected from random bit-errors by both the 802.11 FCS and by the ICV, a valid FCS and ICV but invalid MIC implies that the packet is almost surely a forgery. The MIC protects the source and destination addresses from change, so packets can no longer be redirected to unauthorized destinations or the source spoofed.

On reception, the encapsulation steps are largely reversed. The receiver uses the transmit address and key id to locate the correct set of temporal keys. With this done, the packet sequence number is extracted from the packet and checked for replay. If the packet sequence number has already been seen, the packet is dropped. If this is a new sequence number, TKIP uses the key mixing function to combine the sequence number with the temporal decryption key to produce the WEP per-packet decryption key. This is turned over to the WEP hardware, which decrypts the data field and checks the ICV. If the ICV check fails, then the packet is discarded and a MIB counter incremented. Otherwise, the ICV succeeds; the receiver advances the window of received sequence numbers, and adds the MPDU to any other fragments required to reconstruct the MSDU. Once all the MPDU fragments are assembled into an MSDU, the implementation uses the Michael verification predicate to test for forgeries. If the predicate returns false, TKIP discards the MSDU as a forgery and invokes the countermeasures. Otherwise the packet is delivered upward. The following is a block diagram depicting the TKIP decapsulation process.



Summary

That's all there is to it!

TKIP is a wrapper around WEP and does not constitute an ideal security protocol design. The TKIP MIC is weak enough to require countermeasures to be effective, but together these are strong enough to diminish the likelihood that forgeries succeed. The key mixing function is extra work that would be unnecessary if the original WEP design had selected almost any cipher other than RC4, but preserves the original hardware investment. The long-term cryptographic strength of TKIP is questionable, but removes weak key attacks long enough to deploy new hardware that is capable of a more robust approach. The reuse of the WEP packet formats and IV space forces rekeying at an uncomfortably rapid rate, but this makes a virtue of WEP's poor original design. TKIP will certainly degrade performance at many access points, where it can consume every spare CPU cycle, but removes the worry of trivial compromise.

TKIP seems to meet its entire design goal, which is to provide a tolerable level of security that can be implemented as a software upgrade on legacy hardware. Its design is nearly complete. Its design team includes well-known cryptographers, and, unlike WEP, it has received extensive review both by the broader security community for correctness and by the 802.11 hardware vendors for feasibility. It trades off security to achieve acceptable performance, and it is not the ideal solution. However, TKIP appears to be about the best possible solution within the deployed hardware base.

Although the design of a security protocol is never trivial, it is certainly easier to provide a much more robust solution than TKIP, if the constraint of existing hardware is removed. In the final installment of this series next month, we will examine such a solution being developed within TGi, based on the Advanced Encryption Standard (AES).

For Further Reading

[1] IEEE Std 802.11, Standards for Local and Metropolitan Area Networks: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, 1999.

[2] Ferguson, N., "Michael: an improved MIC for 802.11 WEP," IEEE 802.11 doc 02-020r0, January 17, 2002. Available at http://grouper.ieee.org/groups/802/11/

[3] Stanley, D., "IV Sequencing Requirements Summary," IEEE 802.11 doc 02-006r2, January 18, 2002. Available at http://grouper.ieee.org/groups/802/11/

[4] Housely, R., and D. Whiting, "Temporal Key Hash," IEEE 802.11 doc 01-550r1 October 31, 2001. Available at http://grouper.ieee.org/groups/802/11/

About the Author

Jesse Walker is the network security architect for Intel's Platform Networking Group. He is the technical editor for the 802.11 security enhancements, and was the first person to publicly identify the security flaws in the original 802.11 WLAN protocol. He joined Intel as part of the Shiva acquisition. Jesse has also been active in other industry standardization efforts, including IPSec. Jesse holds a Ph.D. in mathematics from the University of Texas at Austin. Jesse may be contacted at jesse.walker@intel.com.